DPDK

DATA PLANE DEVELOPMENT KIT

DPDK documentation

Release 20.05.0

Aug 03, 2022

CONTENTS

CHAPTER

ONE

GETTING STARTED GUIDE FOR LINUX

1.1 Introduction

This document contains instructions for installing and configuring the Data Plane Development Kit (DPDK) software. It is designed to get customers up and running quickly. The document describes how to compile and run a DPDK application in a Linux application (linux) environment, without going deeply into detail.

1.1.1 Documentation Roadmap

The following is a list of DPDK documents in the suggested reading order:

- Release Notes: Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- Getting Started Guide (this document): Describes how to install and configure the DPDK; designed to get users up and running quickly with the software.
- Programmer's Guide: Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux application (linux) environment
  - The content of the DPDK, the build system (including the commands that can be used in the root DPDK Makefile to build the development kit and an application) and guidelines for porting an application
  - Optimizations used in the software and those that should be considered for new development

  A glossary of terms is also provided.
- API Reference: Provides detailed information about DPDK functions, data structures and other programming constructs.
- Sample Applications User Guide: Describes a set of sample applications. Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

1.2 System Requirements

This chapter describes the packages required to compile the DPDK.

Note: If the DPDK is being used on an Intel® Communications Chipset 89xx Series platform, please consult the *Intel® Communications Chipset 89xx Series Software for Linux Getting Started Guide*.

1.2.1 BIOS Setting Prerequisite on x86

For the majority of platforms, no special BIOS settings are needed to use basic DPDK functionality. However, for additional HPET timer and power management functionality, and high performance of small packets, BIOS setting changes may be needed. Consult the section on *Enabling Additional Functionality* for more information on the required changes.

Note: If UEFI secure boot is enabled, the Linux kernel may disallow the use of UIO on the system. Therefore, devices for use by DPDK should be bound to the `vfio-pci` kernel module rather than `igb_uio` or `uio_pci_generic`. For more details see *Binding and Unbinding Network Ports to/from the Kernel Modules*.

1.2.2 Compilation of the DPDK

Required Tools and Libraries:

Note: The setup commands and installed packages needed on various systems may be different. For details on Linux distributions and the versions tested, please consult the DPDK Release Notes.

- General development tools including `make`, and a supported C compiler such as `gcc` (version 4.9+) or `clang` (version 3.4+).
  - For RHEL/Fedora systems these can be installed using `dnf groupinstall "Development Tools"`
  - For Ubuntu/Debian systems these can be installed using `apt install build-essential`
- Python, recommended version 3.5+.
  - Python v3.5+ is needed to build DPDK using meson and ninja
  - Python 2.7+ or 3.2+, to use various helper scripts included in the DPDK package.
- Meson (version 0.47.1+) and ninja
  - `meson` & `ninja-build` packages in most Linux distributions
  - If the packaged version is below the minimum version, the latest versions can be installed from Python's "pip" repository: `pip3 install meson ninja`
- Library for handling NUMA (Non Uniform Memory Access).
  - `numactl-devel` in RHEL/Fedora;

  - `libnuma-dev` in Debian/Ubuntu;

- Linux kernel headers or sources required to build kernel modules.

Note: Please ensure that the latest patches are applied to third party libraries and software to avoid any known vulnerabilities.

Optional Tools:

- Intel® C++ Compiler (icc). For installation, additional libraries may be required. See the icc Installation Guide found in the Documentation directory under the compiler installation.
- IBM® Advance ToolChain for Powerlinux. This is a set of open source development tools and runtime libraries which allows users to take leading edge advantage of IBM's latest POWER hardware features on Linux. To install it, see the IBM official installation document.

Additional Libraries

A number of DPDK components, such as libraries and poll-mode drivers (PMDs) have additional dependencies. For DPDK builds using meson, the presence or absence of these dependencies will be automatically detected enabling or disabling the relevant components appropriately.

For builds using make, these components are disabled in the default configuration and need to be enabled manually by changing the relevant setting to "y" in the build configuration file i.e. the `.config` file in the build folder.

In each case, the relevant library development package (`-devel` or `-dev`) is needed to build the DPDK components.

For libraries the additional dependencies include:

- libarchive: for some unit tests using tar to get their resources.
- libelf: to compile and use the bpf library.

For poll-mode drivers, the additional dependencies for each driver can be found in that driver's documentation in the relevant DPDK guide document, e.g. *Network Interface Controller Drivers*

1.2.3 Running DPDK Applications

To run an DPDK application, some customization may be required on the target machine.

System Software

Required:

- Kernel version >= 3.16

  The kernel version required is based on the oldest long term stable kernel available at kernel.org when the DPDK version is in development. Compatibility for recent distribution kernels will be kept, notably RHEL/CentOS 7.

  The kernel version in use can be checked using the command:

```
uname -r
```

- glibc >= 2.7 (for features related to cpuset)

  The version can be checked using the `ldd --version` command.

- Kernel configuration

  In the Fedora OS and other common distributions, such as Ubuntu, or Red Hat Enterprise Linux, the vendor supplied kernel configurations can be used to run most DPDK applications.

  For other kernel builds, options which should be enabled for DPDK include:

  - HUGETLBFS
  - PROC_PAGE_MONITOR support
  - HPET and HPET_MMAP configuration options should also be enabled if HPET support is required. See the section on *High Precision Event Timer (HPET) Functionality* for more details.

Use of Hugepages in the Linux Environment

Hugepage support is required for the large memory pool allocation used for packet buffers (the HUGETLBFS option must be enabled in the running kernel as indicated the previous section). By using hugepage allocations, performance is increased since fewer pages are needed, and therefore less Translation Lookaside Buffers (TLBs, high speed translation caches), which reduce the time it takes to translate a virtual page address to a physical page address. Without hugepages, high TLB miss rates would occur with the standard 4k page size, slowing performance.

Reserving Hugepages for DPDK Use

The allocation of hugepages should be done at boot time or as soon as possible after system boot to prevent memory from being fragmented in physical memory. To reserve hugepages at boot time, a parameter is passed to the Linux kernel on the kernel command line.

For 2 MB pages, just pass the hugepages option to the kernel. For example, to reserve 1024 pages of 2 MB, use:

```
hugepages=1024
```

For other hugepage sizes, for example 1G pages, the size must be specified explicitly and can also be optionally set as the default hugepage size for the system. For example, to reserve 4G of hugepage memory in the form of four 1G pages, the following options should be passed to the kernel:

```
default_hugepagesz=1G hugepagesz=1G hugepages=4
```

Note: The hugepage sizes that a CPU supports can be determined from the CPU flags on Intel architecture. If pse exists, 2M hugepages are supported; if pdpe1gb exists, 1G hugepages are supported. On IBM Power architecture, the supported hugepage sizes are 16MB and 16GB.

Note: For 64-bit applications, it is recommended to use 1 GB hugepages if the platform supports them.

In the case of a dual-socket NUMA system, the number of hugepages reserved at boot time is generally divided equally between the two sockets (on the assumption that sufficient memory is present on both sockets).

See the Documentation/admin-guide/kernel-parameters.txt file in your Linux source tree for further details of these and other kernel options.

Alternative:

For 2 MB pages, there is also the option of allocating hugepages after the system has booted. This is done by echoing the number of hugepages required to a nr_hugepages file in the `/sys/devices/` directory. For a single-node system, the command to use is as follows (assuming that 1024 pages are required):

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

On a NUMA machine, pages should be allocated explicitly on separate nodes:

```
echo 1024 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/devices/system/node/node1/hugepages/hugepages-2048kB/nr_hugepages
```

Note: For 1G pages, it is not possible to reserve the hugepage memory after the system has booted.

Using Hugepages with the DPDK

Once the hugepage memory is reserved, to make the memory available for DPDK use, perform the following steps:

```
mkdir /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
```

The mount point can be made permanent across reboots, by adding the following line to the `/etc/fstab` file:

```
nodev /mnt/huge hugetlbfs defaults 0 0
```

For 1GB pages, the page size must be specified as a mount option:

```
nodev /mnt/huge_1GB hugetlbfs pagesize=1GB 0 0
```

1.3 Compiling the DPDK Target from Source

Note: Parts of this process can also be done using the setup script described in the *Quick Start Setup Script* section of this document.

1.3.1 Uncompress DPDK and Browse Sources

First, uncompress the archive and move to the uncompressed DPDK source directory:

```
tar xJf dpdk-<version>.tar.xz
cd dpdk-<version>
```

The DPDK is composed of several directories:

- lib: Source code of DPDK libraries
- drivers: Source code of DPDK poll-mode drivers
- app: Source code of DPDK applications (automatic tests)
- examples: Source code of DPDK application examples
- config, buildtools, mk: Framework-related makefiles, scripts and configuration

1.3.2 Compiling and Installing DPDK System-wide

DPDK can be configured, built and installed on your system using the tools `meson` and `ninja`.

Note: The older makefile-based build system used in older DPDK releases is still present and its use is described in section *Installation of DPDK Target Environment using Make*.

DPDK Configuration

To configure a DPDK build use:

```
meson <options> build
```

where "build" is the desired output build directory, and "<options>" can be empty or one of a number of meson or DPDK-specific build options, described later in this section. The configuration process will finish with a summary of what DPDK libraries and drivers are to be built and installed, and for each item disabled, a reason why that is the case. This information can be used, for example, to identify any missing required packages for a driver.

Once configured, to build and then install DPDK system-wide use:

```
cd build
ninja
ninja install
ldconfig
```

The last two commands above generally need to be run as root, with the *ninja install* step copying the built objects to their final system-wide locations, and the last step causing the dynamic loader *ld.so* to update its cache to take account of the new objects.

Note: On some linux distributions, such as Fedora or Redhat, paths in */usr/local* are not in the default paths for the loader. Therefore, on these distributions, */usr/local/lib* and */usr/local/lib64* should be added to a file in */etc/ld.so.conf.d/* before running *ldconfig*.

Adjusting Build Options

DPDK has a number of options that can be adjusted as part of the build configuration process. These options can be listed by running `meson configure` inside a configured build folder. Many of these options come from the "meson" tool itself and can be seen documented on the Meson Website.

For example, to change the build-type from the default, "debugoptimized", to a regular "debug" build, you can either:

- pass `-Dbuildtype=debug` or `--buildtype=debug` to meson when configuring the build folder initially
- run `meson configure -Dbuildtype=debug` inside the build folder after the initial meson run.

Other options are specific to the DPDK project but can be adjusted similarly. To set the "max_lcores" value to 256, for example, you can either:

- pass `-Dmax_lcores=256` to meson when configuring the build folder initially
- run `meson configure -Dmax_lcores=256` inside the build folder after the initial meson run.

Some of the DPDK sample applications in the *examples* directory can be automatically built as part of a meson build too. To do so, pass a comma-separated list of the examples to build to the *-Dexamples* meson option as below:

```
meson -Dexamples=l2fwd,l3fwd build
```

As with other meson options, this can also be set post-initial-config using *meson configure* in the build directory. There is also a special value "all" to request that all example applications whose dependencies are met on the current system are built. When *-Dexamples=all* is set as a meson option, meson will check each example application to see if it can be built, and add all which can be built to the list of tasks in the ninja build configuration file.

Building Applications Using Installed DPDK

When installed system-wide, DPDK provides a pkg-config file `libdpdk.pc` for applications to query as part of their build. It's recommended that the pkg-config file be used, rather than hard-coding the parameters (cflags/ldflags) for DPDK into the application build process.

An example of how to query and use the pkg-config file can be found in the `Makefile` of each of the example applications included with DPDK. A simplified example snippet is shown below, where the target binary name has been stored in the variable `$(APP)` and the sources for that build are stored in `$(SRCS-y)`.

```
PKGCONF = pkg-config

CFLAGS += -O3 $(shell $(PKGCONF) --cflags libdpdk)
LDFLAGS += $(shell $(PKGCONF) --libs libdpdk)

$(APP): $(SRCS-y) Makefile
        $(CC) $(CFLAGS) $(SRCS-y) -o $@ $(LDFLAGS)
```

Note: Unlike with the older make build system, the meson system is not designed to be used directly from a build directory. Instead it is recommended that it be installed either system-wide or to a known

location in the user's home directory. The install location can be set using the *–prefix* meson option (default: */usr/local*).

an equivalent build recipe for a simple DPDK application using meson as a build system is shown below:

```
project('dpdk-app', 'c')

dpdk = dependency('libdpdk')
sources = files('main.c')
executable('dpdk-app', sources, dependencies: dpdk)
```

1.3.3 Installation of DPDK Target Environment using Make

Note: The building of DPDK using make will be deprecated in a future release. It is therefore recommended that DPDK installation is done using meson and ninja as described above.

The format of a DPDK target is:

```
ARCH-MACHINE-EXECENV-TOOLCHAIN
```

where:

- ARCH can be: i686, x86_64, ppc_64, arm64
- MACHINE can be: native, power8, armv8a
- EXECENV can be: linux, freebsd
- TOOLCHAIN can be: gcc, icc

The targets to be installed depend on the 32-bit and/or 64-bit packages and compilers installed on the host. Available targets can be found in the DPDK/config directory. The defconfig_ prefix should not be used.

Note: Configuration files are provided with the RTE_MACHINE optimization level set. Within the configuration files, the RTE_MACHINE configuration value is set to native, which means that the compiled software is tuned for the platform on which it is built. For more information on this setting, and its possible values, see the *DPDK Programmers Guide*.

When using the Intel® C++ Compiler (icc), one of the following commands should be invoked for 64-bit or 32-bit use respectively. Notice that the shell scripts update the $PATH variable and therefore should not be performed in the same session. Also, verify the compiler's installation directory since the path may be different:

```
source /opt/intel/bin/iccvars.sh intel64
source /opt/intel/bin/iccvars.sh ia32
```

To install and make targets, use the `make install T=<target>` command in the top-level DPDK directory.

For example, to compile a 64-bit target using icc, run:

```
make install T=x86_64-native-linux-icc
```

To compile a 32-bit build using gcc, the make command should be:

```
make install T=i686-native-linux-gcc
```

To prepare a target without building it, for example, if the configuration changes need to be made before compilation, use the `make config T=<target>` command:

```
make config T=x86_64-native-linux-gcc
```

Warning: Any kernel modules to be used, e.g. `igb_uio`, `kni`, must be compiled with the same kernel as the one running on the target. If the DPDK is not being built on the target machine, the `RTE_KERNELDIR` environment variable should be used to point the compilation at a copy of the kernel version to be used on the target machine.

Once the target environment is created, the user may move to the target environment directory and continue to make code changes and re-compile. The user may also make modifications to the compile-time DPDK configuration by editing the .config file in the build directory. (This is a build-local copy of the defconfig file from the top- level config directory).

```
cd x86_64-native-linux-gcc
vi .config
make
```

In addition, the make clean command can be used to remove any existing compiled files for a subsequent full, clean rebuild of the code.

1.3.4 Browsing the Installed DPDK Environment Target

Once a target is created it contains all libraries, including poll-mode drivers, and header files for the DPDK environment that are required to build customer applications. In addition, the test and testpmd applications are built under the build/app directory, which may be used for testing. A kmod directory is also present that contains kernel modules which may be loaded if needed.

1.4 Cross compile DPDK for ARM64

This chapter describes how to cross compile DPDK for ARM64 from x86 build hosts.

Note: Whilst it is recommended to natively build DPDK on ARM64 (just like with x86), it is also possible to cross-build DPDK for ARM64. An ARM64 cross compile GNU toolchain is used for this.

1.4.1 Obtain the cross tool chain

The latest cross compile tool chain can be downloaded from: https://developer.arm.com/open-source/gnu-toolchain/gnu-a/downloads.

It is always recommended to check and get the latest compiler tool from the page and use it to generate better code. As of this writing 8.3-2019.03 is the newest, the following description is an example of this version.

```
wget https://developer.arm.com/-/media/Files/downloads/gnu-a/8.3-2019.03/binrel/gcc-arm-8.3-
↪2019.03-x86_64-aarch64-linux-gnu.tar.xz
```

1.4.2 Unzip and add into the PATH

```
tar -xvf gcc-arm-8.3-2019.03-x86_64-aarch64-linux-gnu.tar.xz
export PATH=$PATH:<cross_install_dir>/gcc-arm-8.3-2019.03-x86_64-aarch64-linux-gnu/bin
```

Note: For the host requirements and other info, refer to the release note section: https://releases.linaro.org/components/toolchain/binaries/

1.4.3 Getting the prerequisite library

NUMA is required by most modern machines, not needed for non-NUMA architectures.

Note: For compiling the NUMA lib, run libtool –version to ensure the libtool version >= 2.2, otherwise the compilation will fail with errors.

```
git clone https://github.com/numactl/numactl.git
cd numactl
git checkout v2.0.13 -b v2.0.13
./autogen.sh
autoconf -i
./configure --host=aarch64-linux-gnu CC=aarch64-linux-gnu-gcc --prefix=<numa install dir>
make install
```

The numa header files and lib file is generated in the include and lib folder respectively under <numa install dir>.

1.4.4 Augment the cross toolchain with NUMA support

Note: This way is optional, an alternative is to use extra CFLAGS and LDFLAGS, depicted in *Cross Compiling DPDK using Meson* below.

Copy the NUMA header files and lib to the cross compiler's directories:

```
cp <numa_install_dir>/include/numa*.h <cross_install_dir>/gcc-arm-8.3-2019.03-x86_64-aarch64-
↪linux-gnu/aarch64-linux-gnu/libc/usr/include/
cp <numa_install_dir>/lib/libnuma.a <cross_install_dir>/gcc-arm-8.3-2019.03-x86_64-aarch64-
↪linux-gnu/lib/gcc/aarch64-linux-gnu/8.3.0/
cp <numa_install_dir>/lib/libnuma.so <cross_install_dir>/gcc-arm-8.3-2019.03-x86_64-aarch64-
↪linux-gnu/lib/gcc/aarch64-linux-gnu/8.3.0/
```

1.4.5 Cross Compiling DPDK using Meson

Meson depends on pkgconfig to find the dependencies. The package `pkg-config-aarch64-linux-gnu` is required for aarch64. To install it in Ubuntu:

```
sudo apt-get install pkg-config-aarch64-linux-gnu
```

To cross-compile DPDK on a desired target machine we can use the following command:

```
meson cross-build --cross-file <target_machine_configuration>
ninja -C cross-build
```

For example if the target machine is arm64 we can use the following command:

```
meson arm64-build --cross-file config/arm/arm64_armv8_linux_gcc
ninja -C arm64-build
```

1.4.6 Configure and Cross Compile DPDK using Make

To configure a build, choose one of the target configurations, like arm64-dpaa-linux-gcc and arm64-thunderx-linux-gcc.

```
make config T=arm64-armv8a-linux-gcc
```

To cross-compile, without compiling the kernel modules, use the following command:

```
make -j CROSS=aarch64-linux-gnu- CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n
```

To cross-compile, including the kernel modules, the kernel source tree needs to be specified by setting RTE_KERNELDIR:

```
make -j CROSS=aarch64-linux-gnu- RTE_KERNELDIR=<kernel_src_rootdir> CROSS_COMPILE=aarch64-
↪linux-gnu-
```

To compile for non-NUMA targets, without compiling the kernel modules, use the following command:

```
make -j CROSS=aarch64-linux-gnu- CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n CONFIG_RTE_
↪LIBRTE_VHOST_NUMA=n CONFIG_RTE_EAL_NUMA_AWARE_HUGEPAGES=n
```

Note: 1. EXTRA_CFLAGS and EXTRA_LDFLAGS should be added to include the NUMA headers and link the library respectively, if the above step *Augment the cross toolchain with NUMA support* was skipped therefore the toolchain was not augmented with NUMA support.

2. "-isystem <numa_install_dir>/include" should be add to EXTRA_CFLAGS, otherwise the numa.h file will get a lot of compiling errors of Werror=cast-qual, Werror=strict-prototypes and Werror=old-style-definition.

An example is given below:

```
make -j CROSS=aarch64-linux-gnu- CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n EXTRA_CFLAGS="-
↪isystem <numa_install_dir>/include" EXTRA_LDFLAGS="-L<numa_install_dir>/lib -lnuma"
```

1.5 Linux Drivers

Different PMDs may require different kernel drivers in order to work properly. Depends on the PMD being used, a corresponding kernel driver should be load and bind to the network ports.

1.5.1 UIO

A small kernel module to set up the device, map device memory to user-space and register interrupts. In many cases, the standard `uio_pci_generic` module included in the Linux kernel can provide the uio capability. This module can be loaded using the command:

```
sudo modprobe uio_pci_generic
```

Note: `uio_pci_generic` module doesn't support the creation of virtual functions.

As an alternative to the `uio_pci_generic`, the DPDK also includes the igb_uio module which can be found in the kmod subdirectory referred to above. It can be loaded as shown below:

```
sudo modprobe uio
sudo insmod kmod/igb_uio.ko
```

Note: `igb_uio` module is disabled by default starting from `DPDK v20.02`. To build it, the config option `CONFIG_RTE_EAL_IGB_UIO` should be enabled. It is planned to move `igb_uio` module to a different git repository.

Note: For some devices which lack support for legacy interrupts, e.g. virtual function (VF) devices, the `igb_uio` module may be needed in place of `uio_pci_generic`.

Note: If UEFI secure boot is enabled, the Linux kernel may disallow the use of UIO on the system. Therefore, devices for use by DPDK should be bound to the `vfio-pci` kernel module rather than `igb_uio` or `uio_pci_generic`. For more details see *Binding and Unbinding Network Ports to/from the Kernel Modules* below.

Note: If the devices used for DPDK are bound to the `uio_pci_generic` kernel module, please make sure that the IOMMU is disabled or passthrough. One can add `intel_iommu=off` or `amd_iommu=off` or `intel_iommu=on iommu=pt``in GRUB command line on x86_64 systems, or add ``iommu.passthrough=1` on arm64 system.

Since DPDK release 1.7 onward provides VFIO support, use of UIO is optional for platforms that support using VFIO.

1.5.2 VFIO

A more robust and secure driver in compare to the `UIO`, relying on IOMMU protection. To make use of VFIO, the `vfio-pci` module must be loaded:

```
sudo modprobe vfio-pci
```

Note that in order to use VFIO, your kernel must support it. VFIO kernel modules have been included in the Linux kernel since version 3.6.0 and are usually present by default, however please consult your distributions documentation to make sure that is the case.

Also, to use VFIO, both kernel and BIOS must support and be configured to use IO virtualization (such as Intel® VT-d).

Note: `vfio-pci` module doesn't support the creation of virtual functions.

For proper operation of VFIO when running DPDK applications as a non-privileged user, correct permissions should also be set up. This can be done by using the DPDK setup script (called dpdk-setup.sh and located in the usertools directory).

Note: VFIO can be used without IOMMU. While this is just as unsafe as using UIO, it does make it possible for the user to keep the degree of device access and programming that VFIO has, in situations where IOMMU is not available.

1.5.3 Bifurcated Driver

PMDs which use the bifurcated driver co-exists with the device kernel driver. On such model the NIC is controlled by the kernel, while the data path is performed by the PMD directly on top of the device.

Such model has the following benefits:

- It is secure and robust, as the memory management and isolation is done by the kernel.
- It enables the user to use legacy linux tools such as `ethtool` or `ifconfig` while running DPDK application on the same network ports.
- It enables the DPDK application to filter only part of the traffic, while the rest will be directed and handled by the kernel driver. The flow bifurcation is performed by the NIC hardware. As an example, using *Flow isolated mode* allows to choose strictly what is received in DPDK.

More about the bifurcated driver can be found in Mellanox Bifurcated DPDK PMD.

1.5.4 Binding and Unbinding Network Ports to/from the Kernel Modules

Note: PMDs Which use the bifurcated driver should not be unbind from their kernel drivers. this section is for PMDs which use the UIO or VFIO drivers.

As of release 1.4, DPDK applications no longer automatically unbind all supported network ports from the kernel driver in use. Instead, in case the PMD being used use the UIO or VFIO drivers, all ports that are to be used by an DPDK application must be bound to the `uio_pci_generic`, `igb_uio` or `vfio-pci` module before the application is run. For such PMDs, any network ports under Linux* control will be ignored and cannot be used by the application.

To bind ports to the `uio_pci_generic`, `igb_uio` or `vfio-pci` module for DPDK use, and then subsequently return ports to Linux* control, a utility script called dpdk-devbind.py is provided in the usertools subdirectory. This utility can be used to provide a view of the current state of the network ports on the system, and to bind and unbind those ports from the different kernel modules, including the uio and vfio modules. The following are some examples of how the script can be used. A full description of the script and its parameters can be obtained by calling the script with the `--help` or `--usage` options. Note that the uio or vfio kernel modules to be used, should be loaded into the kernel before running the `dpdk-devbind.py` script.

Warning: Due to the way VFIO works, there are certain limitations to which devices can be used with VFIO. Mainly it comes down to how IOMMU groups work. Any Virtual Function device can be used with VFIO on its own, but physical devices will require either all ports bound to VFIO, or some of them bound to VFIO while others not being bound to anything at all.

If your device is behind a PCI-to-PCI bridge, the bridge will then be part of the IOMMU group in which your device is in. Therefore, the bridge driver should also be unbound from the bridge PCI device for VFIO to work with devices behind the bridge.

Warning: While any user can run the dpdk-devbind.py script to view the status of the network ports, binding or unbinding network ports requires root privileges.

To see the status of all network ports on the system:

```
./usertools/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
============================================
0000:82:00.0 '82599EB 10-GbE NIC' drv=uio_pci_generic unused=ixgbe
0000:82:00.1 '82599EB 10-GbE NIC' drv=uio_pci_generic unused=ixgbe

Network devices using kernel driver
===================================
0000:04:00.0 'I350 1-GbE NIC' if=em0  drv=igb unused=uio_pci_generic *Active*
0000:04:00.1 'I350 1-GbE NIC' if=eth1 drv=igb unused=uio_pci_generic
0000:04:00.2 'I350 1-GbE NIC' if=eth2 drv=igb unused=uio_pci_generic
0000:04:00.3 'I350 1-GbE NIC' if=eth3 drv=igb unused=uio_pci_generic

Other network devices
=====================
<none>
```

To bind device `eth1`,``04:00.1``, to the `uio_pci_generic` driver:

```
./usertools/dpdk-devbind.py --bind=uio_pci_generic 04:00.1
```

or, alternatively,

```
./usertools/dpdk-devbind.py --bind=uio_pci_generic eth1
```

To restore device `82:00.0` to its original kernel binding:

```
./usertools/dpdk-devbind.py --bind=ixgbe 82:00.0
```

1.6 Compiling and Running Sample Applications

The chapter describes how to compile and run applications in an DPDK environment. It also provides a pointer to where sample applications are stored.

Note: Parts of this process can also be done using the setup script described the *Quick Start Setup Script* section of this document.

1.6.1 Compiling a Sample Application

Once an DPDK target environment directory has been created (such as `x86_64-native-linux-gcc`), it contains all libraries and header files required to build an application.

When compiling an application in the Linux* environment on the DPDK, the following variables must be exported:

- `RTE_SDK` - Points to the DPDK installation directory.
- `RTE_TARGET` - Points to the DPDK target environment directory.

The following is an example of creating the `helloworld` application, which runs in the DPDK Linux environment. This example may be found in the `${RTE_SDK}/examples` directory.

The directory contains the `main.c` file. This file, when combined with the libraries in the DPDK target environment, calls the various functions to initialize the DPDK environment, then launches an entry point (dispatch application) for each core to be utilized. By default, the binary is generated in the build directory.

```
cd examples/helloworld/
export RTE_SDK=$HOME/DPDK
export RTE_TARGET=x86_64-native-linux-gcc

make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map

ls build/app
    helloworld helloworld.map
```

Note: In the above example, `helloworld` was in the directory structure of the DPDK. However, it could have been located outside the directory structure to keep the DPDK structure intact. In the following case, the `helloworld` application is copied to a new directory as a new starting point.

```
export RTE_SDK=/home/user/DPDK
cp -r $(RTE_SDK)/examples/helloworld my_rte_app
cd my_rte_app/
export RTE_TARGET=x86_64-native-linux-gcc

make
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map
```

1.6.2 Running a Sample Application

Warning: Before running the application make sure:

- Hugepages setup is done.
- Any kernel driver being used is loaded.
- In case needed, ports being used by the application should be bound to the corresponding kernel driver.

refer to *Linux Drivers* for more details.

The application is linked with the DPDK target environment's Environmental Abstraction Layer (EAL) library, which provides some options that are generic to every DPDK application.

The following is the list of options that can be given to the EAL:

```
./rte-app [-c COREMASK | -l CORELIST] [-n NUM] [-b <domain:bus:devid.func>] \
          [--socket-mem=MB,...] [-d LIB.so|DIR] [-m MB] [-r NUM] [-v] [--file-prefix] \
          [--proc-type <primary|secondary|auto>]
```

The EAL options are as follows:

- `-c COREMASK` or `-l CORELIST`: An hexadecimal bit mask of the cores to run on. Note that core numbering can change between platforms and should be determined beforehand. The corelist is a set of core numbers instead of a bitmap core mask.
- `-n NUM`: Number of memory channels per processor socket.
- `-b <domain:bus:devid.func>`: Blacklisting of ports; prevent EAL from using specified PCI device (multiple `-b` options are allowed).
- `--use-device`: use the specified Ethernet device(s) only. Use comma-separate `[domain:]bus:devid.func` values. Cannot be used with `-b` option.
- `--socket-mem`: Memory to allocate from hugepages on specific sockets. In dynamic memory mode, this memory will also be pinned (i.e. not released back to the system until application closes).

- `--socket-limit`: Limit maximum memory available for allocation on each socket. Does not support legacy memory mode.
- `-d`: Add a driver or driver directory to be loaded. The application should use this option to load the pmd drivers that are built as shared libraries.
- `-m MB`: Memory to allocate from hugepages, regardless of processor socket. It is recommended that `--socket-mem` be used instead of this option.
- `-r NUM`: Number of memory ranks.
- `-v`: Display version information on startup.
- `--huge-dir`: The directory where hugetlbfs is mounted.
- `mbuf-pool-ops-name`: Pool ops name for mbuf to use.
- `--file-prefix`: The prefix text used for hugepage filenames.
- `--proc-type`: The type of process instance.
- `--vmware-tsc-map`: Use VMware TSC map instead of native RDTSC.
- `--base-virtaddr`: Specify base virtual address.
- `--vfio-intr`: Specify interrupt type to be used by VFIO (has no effect if VFIO is not used).
- `--legacy-mem`: Run DPDK in legacy memory mode (disable memory reserve/unreserve at runtime, but provide more IOVA-contiguous memory).
- `--single-file-segments`: Store memory segments in fewer files (dynamic memory mode only - does not affect legacy memory mode).

The `-c` or `-l` and option is mandatory; the others are optional.

Copy the DPDK application binary to your target, then run the application as follows (assuming the platform has four memory channels per processor socket, and that cores 0-3 are present and are to be used for running the application):

```
./helloworld -l 0-3 -n 4
```

Note: The `--proc-type` and `--file-prefix` EAL options are used for running multiple DPDK processes. See the "Multi-process Sample Application" chapter in the *DPDK Sample Applications User Guide* and the *DPDK Programmers Guide* for more details.

Logical Core Use by Applications

The coremask (-c 0x0f) or corelist (-l 0-3) parameter is always mandatory for DPDK applications. Each bit of the mask corresponds to the equivalent logical core number as reported by Linux. The preferred corelist option is a cleaner method to define cores to be used. Since these logical core numbers, and their mapping to specific cores on specific NUMA sockets, can vary from platform to platform, it is recommended that the core layout for each platform be considered when choosing the coremask/corelist to use in each case.

On initialization of the EAL layer by an DPDK application, the logical cores to be used and their socket location are displayed. This information can also be determined for all cores on the system by examining the `/proc/cpuinfo` file, for example, by running cat `/proc/cpuinfo`. The physical id attribute listed

for each processor indicates the CPU socket to which it belongs. This can be useful when using other processors to understand the mapping of the logical cores to the sockets.

Note: A more graphical view of the logical core layout may be obtained using the `lstopo` Linux utility. On Fedora Linux, this may be installed and run using the following command:

```
sudo yum install hwloc
./lstopo
```

Warning: The logical core layout can change between different board layouts and should be checked before selecting an application coremask/corelist.

Hugepage Memory Use by Applications

When running an application, it is recommended to use the same amount of memory as that allocated for hugepages. This is done automatically by the DPDK application at startup, if no `-m` or `--socket-mem` parameter is passed to it when run.

If more memory is requested by explicitly passing a `-m` or `--socket-mem` value, the application fails. However, the application itself can also fail if the user requests less memory than the reserved amount of hugepage-memory, particularly if using the `-m` option. The reason is as follows. Suppose the system has 1024 reserved 2 MB pages in socket 0 and 1024 in socket 1. If the user requests 128 MB of memory, the 64 pages may not match the constraints:

- The hugepage memory by be given to the application by the kernel in socket 1 only. In this case, if the application attempts to create an object, such as a ring or memory pool in socket 0, it fails. To avoid this issue, it is recommended that the `--socket-mem` option be used instead of the `-m` option.
- These pages can be located anywhere in physical memory, and, although the DPDK EAL will attempt to allocate memory in contiguous blocks, it is possible that the pages will not be contiguous. In this case, the application is not able to allocate big memory pools.

The socket-mem option can be used to request specific amounts of memory for specific sockets. This is accomplished by supplying the `--socket-mem` flag followed by amounts of memory requested on each socket, for example, supply `--socket-mem=0,512` to try and reserve 512 MB for socket 1 only. Similarly, on a four socket system, to allocate 1 GB memory on each of sockets 0 and 2 only, the parameter `--socket-mem=1024,0,1024` can be used. No memory will be reserved on any CPU socket that is not explicitly referenced, for example, socket 3 in this case. If the DPDK cannot allocate enough memory on each socket, the EAL initialization fails.

1.6.3 Additional Sample Applications

Additional sample applications are included in the ${RTE_SDK}/examples directory. These sample applications may be built and run in a manner similar to that described in earlier sections in this manual. In addition, see the *DPDK Sample Applications User Guide* for a description of the application, specific instructions on compilation and execution and some explanation of the code.

1.6.4 Additional Test Applications

In addition, there are two other applications that are built when the libraries are created. The source files for these are in the DPDK/app directory and are called test and testpmd. Once the libraries are created, they can be found in the build/app directory.

- The test application provides a variety of specific tests for the various functions in the DPDK.
- The testpmd application provides a number of different packet throughput tests and examples of features such as how to use the Flow Director found in the Intel® 82599 10 Gigabit Ethernet Controller.

1.7 EAL parameters

This document contains a list of all EAL parameters. These parameters can be used by any DPDK application running on Linux.

1.7.1 Common EAL parameters

The following EAL parameters are common to all platforms supported by DPDK.

Lcore-related options

- `-c <core mask>`

  Set the hexadecimal bitmask of the cores to run on.

- `-l <core list>`

  List of cores to run on

  The argument format is `<c1>[-c2][,c3[-c4],...]` where `c1`, `c2`, etc are core indexes between 0 and 128.

- `--lcores <core map>`

  Map lcore set to physical cpu set

  The argument format is:

  ```
  <lcores[@cpus]>[<,lcores[@cpus]>...]
  ```

  Lcore and CPU lists are grouped by `(` and `)` Within the group. The `-` character is used as a range separator and `,` is used as a single number separator. The grouping `()` can be omitted for single element group. The `@` can be omitted if cpus and lcores have the same value.

Note: At a given instance only one core option `--lcores`, `-l` or `-c` can be used.

- `--master-lcore <core ID>`

  Core ID that is used as master.

- `-s <service core mask>`

  Hexadecimal bitmask of cores to be used as service cores.

Device-related options

- `-b, --pci-blacklist <[domain:]bus:devid.func>`

  Blacklist a PCI device to prevent EAL from using it. Multiple -b options are allowed.

Note: PCI blacklist cannot be used with `-w` option.

- `-w, --pci-whitelist <[domain:]bus:devid.func>`

  Add a PCI device in white list.

Note: PCI whitelist cannot be used with `-b` option.

- `--vdev <device arguments>`

  Add a virtual device using the format:

  ```
  <driver><id>[,key=val, ...]
  ```

  For example:

  ```
  --vdev 'net_pcap0,rx_pcap=input.pcap,tx_pcap=output.pcap'
  ```

- `-d <path to shared object or directory>`

  Load external drivers. An argument can be a single shared object file, or a directory containing multiple driver shared objects. Multiple -d options are allowed.

- `--no-pci`

  Disable PCI bus.

Multiprocessing-related options

- `--proc-type <primary|secondary|auto>`

  Set the type of the current process.

- `--base-virtaddr <address>`

  Attempt to use a different starting address for all memory maps of the primary DPDK process. This can be helpful if secondary processes cannot start due to conflicts in address map.

Memory-related options

- `-n <number of channels>`

  Set the number of memory channels to use.

- `-r <number of ranks>`

  Set the number of memory ranks (auto-detected by default).

- `-m <megabytes>`

  Amount of memory to preallocate at startup.

- `--in-memory`

  Do not create any shared data structures and run entirely in memory. Implies `--no-shconf` and (if applicable) `--huge-unlink`.

- `--iova-mode <pa|va>`

  Force IOVA mode to a specific value.

Debugging options

- `--no-shconf`

  No shared files created (implies no secondary process support).

- `--no-huge`

  Use anonymous memory instead of hugepages (implies no secondary process support).

- `--log-level <type:val>`

  Specify log level for a specific component. For example:

```
--log-level lib.eal:debug
```

  Can be specified multiple times.

- `--trace=<regex-match>`

  Enable trace based on regular expression trace name. By default, the trace is disabled. User must specify this option to enable trace. For example:

  Global trace configuration for EAL only:

```
--trace=eal
```

Global trace configuration for ALL the components:

```
--trace=.*
```

Can be specified multiple times up to 32 times.

- `--trace-dir=<directory path>`

  Specify trace directory for trace output. For example:

  Configuring `/tmp/` as a trace output directory:

  ```
  --trace-dir=/tmp
  ```

  By default, trace output will created at `home` directory and parameter must be specified once only.

- `--trace-bufsz=<val>`

  Specify maximum size of allocated memory for trace output for each thread. Valid unit can be either `B` or `K` or `M` for `Bytes`, `KBytes` and `MBytes` respectively. For example:

  Configuring `2MB` as a maximum size for trace output file:

  ```
  --trace-bufsz=2M
  ```

  By default, size of trace output file is `1MB` and parameter must be specified once only.

- `--trace-mode=<o[verwrite] | d[iscard] >`

  Specify the mode of update of trace output file. Either update on a file can be wrapped or discarded when file size reaches its maximum limit. For example:

  To `discard` update on trace output file:

  ```
  --trace-mode=d or --trace-mode=discard
  ```

  Default mode is `overwrite` and parameter must be specified once only.

Other options

- `-h`, `--help`

  Display help message listing all EAL parameters.

- `-v`

  Display the version information on startup.

- `mbuf-pool-ops-name`:

  Pool ops name for mbuf to use.

- `--telemetry`:

  Enable telemetry (enabled by default).

- `--no-telemetry`:

  Disable telemetry.

1.7.2 Linux-specific EAL parameters

In addition to common EAL parameters, there are also Linux-specific EAL parameters.

Device-related options

- `--create-uio-dev`

  Create `/dev/uioX` files for devices bound to igb_uio kernel driver (usually done by the igb_uio driver itself).

- `--vmware-tsc-map`

  Use VMware TSC map instead of native RDTSC.

- `--no-hpet`

  Do not use the HPET timer.

- `--vfio-intr <legacy|msi|msix>`

  Use specified interrupt mode for devices bound to VFIO kernel driver.

Multiprocessing-related options

- `--file-prefix <prefix name>`

  Use a different shared data file prefix for a DPDK process. This option allows running multiple independent DPDK primary/secondary processes under different prefixes.

Memory-related options

- `--legacy-mem`

  Use legacy DPDK memory allocation mode.

- `--socket-mem <amounts of memory per socket>`

  Preallocate specified amounts of memory per socket. The parameter is a comma-separated list of values. For example:

  ```
  --socket-mem 1024,2048
  ```

  This will allocate 1 gigabyte of memory on socket 0, and 2048 megabytes of memory on socket 1.

- `--socket-limit <amounts of memory per socket>`

  Place a per-socket upper limit on memory use (non-legacy memory mode only). 0 will disable the limit for a particular socket.

- `--single-file-segments`

  Create fewer files in hugetlbfs (non-legacy mode only).

- `--huge-dir <path to hugetlbfs directory>`

  Use specified hugetlbfs directory instead of autodetected ones.

- `--huge-unlink`

  Unlink hugepage files after creating them (implies no secondary process support).

- `--match-allocations`

  Free hugepages back to system exactly as they were originally allocated.

Other options

- `--syslog <syslog facility>`

  Set syslog facility. Valid syslog facilities are:

```
auth
cron
daemon
ftp
kern
lpr
mail
news
syslog
user
uucp
local0
local1
local2
local3
local4
local5
local6
local7
```

1.8 Enabling Additional Functionality

1.8.1 High Precision Event Timer (HPET) Functionality

BIOS Support

The High Precision Timer (HPET) must be enabled in the platform BIOS if the HPET is to be used. Otherwise, the Time Stamp Counter (TSC) is used by default. The BIOS is typically accessed by pressing F2 while the platform is starting up. The user can then navigate to the HPET option. On the Crystal Forest platform BIOS, the path is: **Advanced -> PCH-IO Configuration -> High Precision Timer ->** (Change from Disabled to Enabled if necessary).

On a system that has already booted, the following command can be issued to check if HPET is enabled:

```
grep hpet /proc/timer_list
```

If no entries are returned, HPET must be enabled in the BIOS (as per the instructions above) and the system rebooted.

Linux Kernel Support

The DPDK makes use of the platform HPET timer by mapping the timer counter into the process address space, and as such, requires that the HPET_MMAP kernel configuration option be enabled.

> **Warning:** On Fedora, and other common distributions such as Ubuntu, the HPET_MMAP kernel option is not enabled by default. To recompile the Linux kernel with this option enabled, please consult the distributions documentation for the relevant instructions.

Enabling HPET in the DPDK

By default, HPET support is disabled in the DPDK build configuration files. To use HPET, the CONFIG_RTE_LIBEAL_USE_HPET setting should be changed to y, which will enable the HPET settings at compile time.

For an application to use the rte_get_hpet_cycles() and rte_get_hpet_hz() API calls, and optionally to make the HPET the default time source for the rte_timer library, the new rte_eal_hpet_init() API call should be called at application initialization. This API call will ensure that the HPET is accessible, returning an error to the application if it is not, for example, if HPET_MMAP is not enabled in the kernel. The application can then determine what action to take, if any, if the HPET is not available at run-time.

Note: For applications that require timing APIs, but not the HPET timer specifically, it is recommended that the rte_get_timer_cycles() and rte_get_timer_hz() API calls be used instead of the HPET-specific APIs. These generic APIs can work with either TSC or HPET time sources, depending on what is requested by an application call to rte_eal_hpet_init(), if any, and on what is available on the system at runtime.

1.8.2 Running DPDK Applications Without Root Privileges

Note: The instructions below will allow running DPDK as non-root with older Linux kernel versions. However, since version 4.0, the kernel does not allow unprivileged processes to read the physical address information from the pagemaps file, making it impossible for those processes to use HW devices which require physical addresses

---

Although applications using the DPDK use network ports and other hardware resources directly, with a number of small permission adjustments it is possible to run these applications as a user other than "root". To do so, the ownership, or permissions, on the following Linux file system objects should be adjusted to ensure that the Linux user account being used to run the DPDK application has access to them:

- All directories which serve as hugepage mount points, for example, /mnt/huge
- The userspace-io device files in /dev, for example, /dev/uio0, /dev/uio1, and so on
- The userspace-io sysfs config and resource files, for example for uio0:

```
/sys/class/uio/uio0/device/config
/sys/class/uio/uio0/device/resource*
```

- If the HPET is to be used, `/dev/hpet`

Note: On some Linux installations, `/dev/hugepages` is also a hugepage mount point created by default.

1.8.3 Power Management and Power Saving Functionality

Enhanced Intel SpeedStep® Technology must be enabled in the platform BIOS if the power management feature of DPDK is to be used. Otherwise, the sys file folder `/sys/devices/system/cpu/cpu0/cpufreq` will not exist, and the CPU frequency- based power management cannot be used. Consult the relevant BIOS documentation to determine how these settings can be accessed.

For example, on some Intel reference platform BIOS variants, the path to Enhanced Intel SpeedStep® Technology is:

```
Advanced
  -> Processor Configuration
  -> Enhanced Intel SpeedStep® Tech
```

In addition, C3 and C6 should be enabled as well for power management. The path of C3 and C6 on the same platform BIOS is:

```
Advanced
  -> Processor Configuration
  -> Processor C3 Advanced
  -> Processor Configuration
  -> Processor C6
```

1.8.4 Using Linux Core Isolation to Reduce Context Switches

While the threads used by an DPDK application are pinned to logical cores on the system, it is possible for the Linux scheduler to run other tasks on those cores also. To help prevent additional workloads from running on those cores, it is possible to use the `isolcpus` Linux kernel parameter to isolate them from the general Linux scheduler.

For example, if DPDK applications are to run on logical cores 2, 4 and 6, the following should be added to the kernel parameter list:

```
isolcpus=2,4,6
```

1.8.5 Loading the DPDK KNI Kernel Module

To run the DPDK Kernel NIC Interface (KNI) sample application, an extra kernel module (the kni module) must be loaded into the running kernel. The module is found in the kmod sub-directory of the DPDK target directory. Similar to the loading of the `igb_uio` module, this module should be loaded using the insmod command as shown below (assuming that the current directory is the DPDK target directory):

```
insmod kmod/rte_kni.ko
```

Note: See the "Kernel NIC Interface Sample Application" chapter in the *DPDK Sample Applications User Guide* for more details.

1.8.6 Using Linux IOMMU Pass-Through to Run DPDK with Intel® VT-d

To enable Intel® VT-d in a Linux kernel, a number of kernel configuration options must be set. These include:

- `IOMMU_SUPPORT`
- `IOMMU_API`
- `INTEL_IOMMU`

In addition, to run the DPDK with Intel® VT-d, the `iommu=pt` kernel parameter must be used when using `igb_uio` driver. This results in pass-through of the DMAR (DMA Remapping) lookup in the host. Also, if `INTEL_IOMMU_DEFAULT_ON` is not set in the kernel, the `intel_iommu=on` kernel parameter must be used too. This ensures that the Intel IOMMU is being initialized as expected.

Please note that while using `iommu=pt` is compulsory for `igb_uio driver`, the `vfio-pci` driver can actually work with both `iommu=pt` and `iommu=on`.

1.9 Quick Start Setup Script

The dpdk-setup.sh script, found in the usertools subdirectory, allows the user to perform the following tasks:

- Build the DPDK libraries
- Insert and remove the DPDK IGB_UIO kernel module
- Insert and remove VFIO kernel modules
- Insert and remove the DPDK KNI kernel module
- Create and delete hugepages for NUMA and non-NUMA cases
- View network port status and reserve ports for DPDK application use
- Set up permissions for using VFIO as a non-privileged user
- Run the test and testpmd applications
- Look at hugepages in the meminfo
- List hugepages in `/mnt/huge`

- Remove built DPDK libraries

Once these steps have been completed for one of the EAL targets, the user may compile their own application that links in the EAL libraries to create the DPDK image.

1.9.1 Script Organization

The dpdk-setup.sh script is logically organized into a series of steps that a user performs in sequence. Each step provides a number of options that guide the user to completing the desired task. The following is a brief synopsis of each step.

Step 1: Build DPDK Libraries

Initially, the user must select a DPDK target to choose the correct target type and compiler options to use when building the libraries.

The user must have all libraries, modules, updates and compilers installed in the system prior to this, as described in the earlier chapters in this Getting Started Guide.

Step 2: Setup Environment

The user configures the Linux* environment to support the running of DPDK applications. Hugepages can be set up for NUMA or non-NUMA systems. Any existing hugepages will be removed. The DPDK kernel module that is needed can also be inserted in this step, and network ports may be bound to this module for DPDK application use.

Step 3: Run an Application

The user may run the test application once the other steps have been performed. The test application allows the user to run a series of functional tests for the DPDK. The testpmd application, which supports the receiving and sending of packets, can also be run.

Step 4: Examining the System

This step provides some tools for examining the status of hugepage mappings.

Step 5: System Cleanup

The final step has options for restoring the system to its original state.

1.9.2 Use Cases

The following are some example of how to use the dpdk-setup.sh script. The script should be run using the source command. Some options in the script prompt the user for further data before proceeding.

> **Warning:** The dpdk-setup.sh script should be run with root privileges.

```
source usertools/dpdk-setup.sh

------------------------------------------------------------------------

RTE_SDK exported as /home/user/rte

------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Step 1: Select the DPDK environment to build

------------------------------------------------------------------------------

[1] i686-native-linux-gcc

[2] i686-native-linux-icc

[3] ppc_64-power8-linux-gcc

[4] x86_64-native-freebsd-clang

[5] x86_64-native-freebsd-gcc

[6] x86_64-native-linux-clang

[7] x86_64-native-linux-gcc

[8] x86_64-native-linux-icc

------------------------------------------------------------------------------

Step 2: Setup linux environment

------------------------------------------------------------------------------

[11] Insert IGB UIO module

[12] Insert VFIO module

[13] Insert KNI module

[14] Setup hugepage mappings for non-NUMA systems

[15] Setup hugepage mappings for NUMA systems

[16] Display current Ethernet device settings

[17] Bind Ethernet device to IGB UIO module

[18] Bind Ethernet device to VFIO module

[19] Setup VFIO permissions

------------------------------------------------------------------------------

Step 3: Run test application for linux environment

------------------------------------------------------------------------------

[20] Run test application ($RTE_TARGET/app/test)

[21] Run testpmd application in interactive mode ($RTE_TARGET/app/testpmd)

------------------------------------------------------------------------------

Step 4: Other tools

------------------------------------------------------------------------------

[22] List hugepage info from /proc/meminfo
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------------

Step 5: Uninstall and system cleanup

----------------------------------------------------------------------------

[23] Uninstall all targets

[24] Unbind NICs from IGB UIO driver

[25] Remove IGB UIO module

[26] Remove VFIO module

[27] Remove KNI module

[28] Remove hugepage mappings

[29] Exit Script
```

Option:

The following selection demonstrates the creation of the `x86_64-native-linux-gcc` DPDK library.

```
Option: 9

================== Installing x86_64-native-linux-gcc

Configuration done
== Build lib
...
Build complete
RTE_TARGET exported as x86_64-native-linux-gcc
```

The following selection demonstrates the starting of the DPDK UIO driver.

```
Option: 25

Unloading any existing DPDK UIO module
Loading DPDK UIO module
```

The following selection demonstrates the creation of hugepages in a NUMA system. 1024 2 MByte pages are assigned to each node. The result is that the application should use -m 4096 for starting the application to access both memory areas (this is done automatically if the -m option is not provided).

Note: If prompts are displayed to remove temporary files, type 'y'.

```
Option: 15

Removing currently reserved hugepages
mounting /mnt/huge and removing directory
Input the number of 2MB pages for each node
Example: to have 128MB of hugepages available per node,
enter '64' to reserve 64 * 2MB pages on each node
Number of pages for node0: 1024
Number of pages for node1: 1024
```

(continues on next page)

(continued from previous page)

```
Reserving hugepages
Creating /mnt/huge and mounting as hugetlbfs
```

The following selection demonstrates the launch of the test application to run on a single core.

```
Option: 20

Enter hex bitmask of cores to execute test app on
Example: to execute app on cores 0 to 7, enter 0xff
bitmask: 0x01
Launching app
EAL: coremask set to 1
EAL: Detected lcore 0 on socket 0
...
EAL: Master core 0 is ready (tid=1b2ad720)
RTE>>
```

1.9.3 Applications

Once the user has run the dpdk-setup.sh script, built one of the EAL targets and set up hugepages (if using one of the Linux EAL targets), the user can then move on to building and running their application or one of the examples provided.

The examples in the /examples directory provide a good starting point to gain an understanding of the operation of the DPDK. The following command sequence shows how the helloworld sample application is built and run. As recommended in Section 4.2.1 , "Logical Core Use by Applications", the logical core layout of the platform should be determined when selecting a core mask to use for an application.

```
cd helloworld/
make
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map

sudo ./build/app/helloworld -l 0-3 -n 3
[sudo] password for rte:

EAL: coremask set to f
EAL: Detected lcore 0 as core 0 on socket 0
EAL: Detected lcore 1 as core 0 on socket 1
EAL: Detected lcore 2 as core 1 on socket 0
EAL: Detected lcore 3 as core 1 on socket 1
EAL: Setting up hugepage memory...
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0add800000 (size = 0x200000)
EAL: Ask a virtual area of 0x3d400000 bytes
EAL: Virtual area found at 0x7f0aa0200000 (size = 0x3d400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9fc00000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9f600000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9f000000 (size = 0x400000)
EAL: Ask a virtual area of 0x800000 bytes
EAL: Virtual area found at 0x7f0a9e600000 (size = 0x800000)
EAL: Ask a virtual area of 0x800000 bytes
```

(continues on next page)

(continued from previous page)

```
EAL: Virtual area found at 0x7f0a9dc00000 (size = 0x800000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9d600000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9d000000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9ca00000 (size = 0x400000)
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0a9c600000 (size = 0x200000)
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0a9c200000 (size = 0x200000)
EAL: Ask a virtual area of 0x3fc00000 bytes
EAL: Virtual area found at 0x7f0a5c400000 (size = 0x3fc00000)
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0a5c000000 (size = 0x200000)
EAL: Requesting 1024 pages of size 2MB from socket 0
EAL: Requesting 1024 pages of size 2MB from socket 1
EAL: Master core 0 is ready (tid=de25b700)
EAL: Core 1 is ready (tid=5b7fe700)
EAL: Core 3 is ready (tid=5a7fc700)
EAL: Core 2 is ready (tid=5affd700)
hello from core 1
hello from core 2
hello from core 3
hello from core 0
```

1.10 How to get best performance with NICs on Intel platforms

This document is a step-by-step guide for getting high performance from DPDK applications on Intel platforms.

1.10.1 Hardware and Memory Requirements

For best performance use an Intel Xeon class server system such as Ivy Bridge, Haswell or newer.

Ensure that each memory channel has at least one memory DIMM inserted, and that the memory size for each is at least 4GB. **Note**: this has one of the most direct effects on performance.

You can check the memory configuration using `dmidecode` as follows:

```
dmidecode -t memory | grep Locator

Locator: DIMM_A1
Bank Locator: NODE 1
Locator: DIMM_A2
Bank Locator: NODE 1
Locator: DIMM_B1
Bank Locator: NODE 1
Locator: DIMM_B2
Bank Locator: NODE 1
...
Locator: DIMM_G1
Bank Locator: NODE 2
Locator: DIMM_G2
Bank Locator: NODE 2
Locator: DIMM_H1
```

(continues on next page)

(continued from previous page)

```
Bank Locator: NODE 2
Locator: DIMM_H2
Bank Locator: NODE 2
```

The sample output above shows a total of 8 channels, from `A` to `H`, where each channel has 2 DIMMs.

You can also use `dmidecode` to determine the memory frequency:

```
dmidecode -t memory | grep Speed

Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
Configured Clock Speed: Unknown
Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
...
Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
Configured Clock Speed: Unknown
Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
Configured Clock Speed: Unknown
```

The output shows a speed of 2133 MHz (DDR4) and Unknown (not existing). This aligns with the previous output which showed that each channel has one memory bar.

Network Interface Card Requirements

Use a DPDK supported high end NIC such as the Intel XL710 40GbE.

Make sure each NIC has been flashed the latest version of NVM/firmware.

Use PCIe Gen3 slots, such as Gen3 `x8` or Gen3 `x16` because PCIe Gen2 slots don't provide enough bandwidth for 2 x 10GbE and above. You can use `lspci` to check the speed of a PCI slot using something like the following:

```
lspci -s 03:00.1 -vv | grep LnkSta

LnkSta: Speed 8GT/s, Width x8, TrErr- Train- SlotClk+ DLActive- ...
LnkSta2: Current De-emphasis Level: -6dB, EqualizationComplete+ ...
```

When inserting NICs into PCI slots always check the caption, such as CPU0 or CPU1 to indicate which socket it is connected to.

Care should be take with NUMA. If you are using 2 or more ports from different NICs, it is best to ensure that these NICs are on the same CPU socket. An example of how to determine this is shown further below.

BIOS Settings

The following are some recommendations on BIOS settings. Different platforms will have different BIOS naming so the following is mainly for reference:

1. Establish the steady state for the system, consider reviewing BIOS settings desired for best performance characteristic e.g. optimize for performance or energy efficiency.
2. Match the BIOS settings to the needs of the application you are testing.
3. Typically, **Performance** as the CPU Power and Performance policy is a reasonable starting point.
4. Consider using Turbo Boost to increase the frequency on cores.
5. Disable all virtualization options when you test the physical function of the NIC, and turn on VT-d if you wants to use VFIO.

Linux boot command line

The following are some recommendations on GRUB boot settings:

1. Use the default grub file as a starting point.
2. Reserve 1G huge pages via grub configurations. For example to reserve 8 huge pages of 1G size:

```
default_hugepagesz=1G hugepagesz=1G hugepages=8
```

3. Isolate CPU cores which will be used for DPDK. For example:

```
isolcpus=2,3,4,5,6,7,8
```

4. If it wants to use VFIO, use the following additional grub parameters:

```
iommu=pt intel_iommu=on
```

1.10.2 Configurations before running DPDK

1. Build the DPDK target and reserve huge pages. See the earlier section on *Use of Hugepages in the Linux Environment* for more details.

   The following shell commands may help with building and configuration:

```
# Build DPDK target.
cd dpdk_folder
make install T=x86_64-native-linux-gcc -j

# Get the hugepage size.
awk '/Hugepagesize/ {print $2}' /proc/meminfo

# Get the total huge page numbers.
awk '/HugePages_Total/ {print $2} ' /proc/meminfo

# Unmount the hugepages.
umount `awk '/hugetlbfs/ {print $2}' /proc/mounts`

# Create the hugepage mount folder.
mkdir -p /mnt/huge
```

(continues on next page)

(continued from previous page)

```
# Mount to the specific folder.
mount -t hugetlbfs nodev /mnt/huge
```

2. Check the CPU layout using the DPDK `cpu_layout` utility:

```
cd dpdk_folder

usertools/cpu_layout.py
```

   Or run `lscpu` to check the cores on each socket.

3. Check your NIC id and related socket id:

```
# List all the NICs with PCI address and device IDs.
lspci -nn | grep Eth
```

   For example suppose your output was as follows:

```
82:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
82:00.1 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
85:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
85:00.1 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
```

   Check the PCI device related numa node id:

```
cat /sys/bus/pci/devices/0000\:xx\:00.x/numa_node
```

   Usually `0x:00.x` is on socket 0 and `8x:00.x` is on socket 1. **Note**: To get the best performance, ensure that the core and NICs are in the same socket. In the example above `85:00.0` is on socket 1 and should be used by cores on socket 1 for the best performance.

4. Check which kernel drivers needs to be loaded and whether there is a need to unbind the network ports from their kernel drivers. More details about DPDK setup and Linux kernel requirements see *Compiling the DPDK Target from Source* and *Linux Drivers*.

CHAPTER

TWO

GETTING STARTED GUIDE FOR FREEBSD

2.1 Introduction

This document contains instructions for installing and configuring the Data Plane Development Kit (DPDK) software. It is designed to get customers up and running quickly and describes how to compile and run a DPDK application in a FreeBSD application (freebsd) environment, without going deeply into detail.

For a comprehensive guide to installing and using FreeBSD, the following handbook is available from the FreeBSD Documentation Project: FreeBSD Handbook.

Note: DPDK is now available as part of the FreeBSD ports collection and as a pre-built package. Installing via the ports collection or FreeBSD *pkg* infrastructure is now the recommended way to install DPDK on FreeBSD, and is documented in the next chapter, *Installing DPDK from the Ports Collection*.

2.1.1 Documentation Roadmap

The following is a list of DPDK documents in the suggested reading order:

- **Release Notes** : Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- **Getting Started Guide** (this document): Describes how to install and configure the DPDK; designed to get users up and running quickly with the software.
- **Programmer's Guide**: Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux* application (linux) environment
  - The content of the DPDK, the build system (including the commands that can be used in the root DPDK Makefile to build the development kit and an application) and guidelines for porting an application
  - Optimizations used in the software and those that should be considered for new development

  A glossary of terms is also provided.
- **API Reference**: Provides detailed information about DPDK functions, data structures and other programming constructs.

- **Sample Applications User Guide**: Describes a set of sample applications. Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

2.2 Installing DPDK from the Ports Collection

The easiest way to get up and running with the DPDK on FreeBSD is to install it using the FreeBSD *pkg* utility or from the ports collection. Details of installing applications from packages or the ports collection are documented in the FreeBSD Handbook, chapter Installing Applications: Packages and Ports.

Note: Please ensure that the latest patches are applied to third party libraries and software to avoid any known vulnerabilities.

2.2.1 Installing the DPDK Package for FreeBSD

DPDK can be installed on FreeBSD using the command:

```
pkg install dpdk
```

After the installation of the DPDK package, instructions will be printed on how to install the kernel modules required to use the DPDK. A more complete version of these instructions can be found in the sections *Loading the DPDK contigmem Module* and *Loading the DPDK nic_uio Module*. Normally, lines like those below would be added to the file `/boot/loader.conf`.

```
# Reserve 2 x 1G blocks of contiguous memory using contigmem driver:
hw.contigmem.num_buffers=2
hw.contigmem.buffer_size=1073741824
contigmem_load="YES"

# Identify NIC devices for DPDK apps to use and load nic_uio driver:
hw.nic_uio.bdfs="2:0:0,2:0:1"
nic_uio_load="YES"
```

2.2.2 Installing the DPDK FreeBSD Port

If so desired, the user can install DPDK using the ports collection rather than from a pre-compiled binary package. On a system with the ports collection installed in `/usr/ports`, the DPDK can be installed using the commands:

```
cd /usr/ports/net/dpdk

make install
```

2.2.3 Compiling and Running the Example Applications

When the DPDK has been installed from the ports collection it installs its example applications in `/usr/local/share/dpdk/examples`. These examples can be compiled and run as described in *Compiling and Running Sample Applications*.

Note: DPDK example applications must be complied using *gmake* rather than BSD *make*. To detect the installed DPDK libraries, *pkg-config* should also be installed on the system.

Note: To install a copy of the DPDK compiled using gcc, please download the official DPDK package from https://core.dpdk.org/download/ and install manually using the instructions given in the next chapter, *Compiling the DPDK Target from Source*

An example application can therefore be copied to a user's home directory and compiled and run as below, where we have 2 memory blocks of size 1G reserved via the contigmem module, and 4 NIC ports bound to the nic_uio module:

```
cp -r /usr/local/share/dpdk/examples/helloworld .

cd helloworld/

gmake
cc -O3 -I/usr/local/include -include rte_config.h -march=corei7 -D__BSD_VISIBLE  main.c -o␣
↪build/helloworld-shared  -L/usr/local/lib -lrte_bpf -lrte_flow_classify -lrte_pipeline -lrte_
↪table -lrte_port -lrte_fib -lrte_ipsec -lrte_stack -lrte_security -lrte_sched -lrte_reorder -
↪lrte_rib -lrte_rcu -lrte_rawdev -lrte_pdump -lrte_member -lrte_lpm -lrte_latencystats -lrte_
↪jobstats -lrte_ip_frag -lrte_gso -lrte_gro -lrte_eventdev -lrte_efd -lrte_distributor -lrte_
↪cryptodev -lrte_compressdev -lrte_cfgfile -lrte_bitratestats -lrte_bbdev -lrte_acl -lrte_
↪timer -lrte_hash -lrte_metrics -lrte_cmdline -lrte_pci -lrte_ethdev -lrte_meter -lrte_net -
↪lrte_mbuf -lrte_mempool -lrte_ring -lrte_eal -lrte_kvargs
ln -sf helloworld-shared build/helloworld

sudo ./build/helloworld -l 0-3
EAL: Sysctl reports 8 cpus
EAL: Detected 8 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'PA'
EAL: Contigmem driver has 2 buffers, each of size 1GB
EAL: Mapped memory segment 0 @ 0x1040000000: physaddr:0x180000000, len 1073741824
EAL: Mapped memory segment 1 @ 0x1080000000: physaddr:0x1c0000000, len 1073741824
EAL: PCI device 0000:00:19.0 on NUMA socket 0
EAL:   probe driver: 8086:153b net_e1000_em
EAL:   0000:00:19.0 not managed by UIO driver, skipping
EAL: PCI device 0000:01:00.0 on NUMA socket 0
EAL:   probe driver: 8086:1572 net_i40e
EAL: PCI device 0000:01:00.1 on NUMA socket 0
EAL:   probe driver: 8086:1572 net_i40e
EAL: PCI device 0000:01:00.2 on NUMA socket 0
EAL:   probe driver: 8086:1572 net_i40e
EAL: PCI device 0000:01:00.3 on NUMA socket 0
EAL:   probe driver: 8086:1572 net_i40e
hello from core 1
hello from core 2
hello from core 3
hello from core 0
```

Note: To run a DPDK process as a non-root user, adjust the permissions on the `/dev/contigmem` and `/dev/uio device` nodes as described in section *Running DPDK Applications Without Root Privileges*

Note: For an explanation of the command-line parameters that can be passed to an DPDK application, see section *Running a Sample Application*.

2.3 Compiling the DPDK Target from Source

2.3.1 Prerequisites

The following FreeBSD packages are required to build DPDK:

- meson
- ninja
- pkgconf

These can be installed using (as root):

```
pkg install meson pkgconf
```

To compile the required kernel modules for memory management and working with physical NIC devices, the kernel sources for FreeBSD also need to be installed. If not already present on the system, these can be installed via commands like the following, for FreeBSD 12.1 on x86_64:

```
fetch http://ftp.freebsd.org/pub/FreeBSD/releases/amd64/12.1-RELEASE/src.txz
tar -C / -xJvf src.txz
```

To enable the telemetry library in DPDK, the jansson library also needs to be installed, and can be installed via:

```
pkg install jansson
```

Individual drivers may have additional requirements. Consult the relevant driver guide for any driver-specific requirements of interest.

2.3.2 Building DPDK

The following commands can be used to build and install DPDK on a system. The final, install, step generally needs to be run as root:

```
meson build
cd build
ninja
ninja install
```

This will install the DPDK libraries and drivers to */usr/local/lib* with a pkg-config file *libdpdk.pc* installed to */usr/local/lib/pkgconfig*. The DPDK test applications, such as *dpdk-testpmd* are installed to

/usr/local/bin. To use these applications, it is recommended that the *contigmem* and *nic_uio* kernel modules be loaded first, as described in the next section.

Note: It is recommended that pkg-config be used to query information about the compiler and linker flags needed to build applications against DPDK. In some cases, the path */usr/local/lib/pkgconfig* may not be in the default search paths for *.pc* files, which means that queries for DPDK information may fail. This can be fixed by setting the appropriate path in *PKG_CONFIG_PATH* environment variable.

2.3.3 Loading the DPDK contigmem Module

To run a DPDK application, physically contiguous memory is required. In the absence of non-transparent superpages, the included sources for the contigmem kernel module provides the ability to present contiguous blocks of memory for the DPDK to use. The contigmem module must be loaded into the running kernel before any DPDK is run. Once DPDK is installed on the system, the module can be found in the */boot/modules* directory.

The amount of physically contiguous memory along with the number of physically contiguous blocks to be reserved by the module can be set at runtime prior to module loading using:

```
kenv hw.contigmem.num_buffers=n
kenv hw.contigmem.buffer_size=m
```

The kernel environment variables can also be specified during boot by placing the following in `/boot/loader.conf`:

```
hw.contigmem.num_buffers=n
hw.contigmem.buffer_size=m
```

The variables can be inspected using the following command:

```
sysctl -a hw.contigmem
```

Where n is the number of blocks and m is the size in bytes of each area of contiguous memory. A default of two buffers of size 1073741824 bytes (1 Gigabyte) each is set during module load if they are not specified in the environment.

The module can then be loaded using kldload:

```
kldload contigmem
```

It is advisable to include the loading of the contigmem module during the boot process to avoid issues with potential memory fragmentation during later system up time. This can be achieved by placing lines similar to the following into `/boot/loader.conf`:

```
hw.contigmem.num_buffers=1
hw.contigmem.buffer_size=1073741824
contigmem_load="YES"
```

Note: The contigmem_load directive should be placed after any definitions of `hw.contigmem.num_buffers` and `hw.contigmem.buffer_size` if the default values are not to be used.

An error such as:

```
kldload: can't load ./x86_64-native-freebsd-gcc/kmod/contigmem.ko:
         Exec format error
```

is generally attributed to not having enough contiguous memory available and can be verified via dmesg or /var/log/messages:

```
kernel: contigmalloc failed for buffer <n>
```

To avoid this error, reduce the number of buffers or the buffer size.

2.3.4 Loading the DPDK nic_uio Module

After loading the contigmem module, the nic_uio module must also be loaded into the running kernel prior to running any DPDK application, e.g. using:

```
kldload nic_uio
```

Note: If the ports to be used are currently bound to a existing kernel driver then the hw.nic_uio.bdfs sysctl value will need to be set before loading the module. Setting this value is described in the next section below.

Currently loaded modules can be seen by using the kldstat command and a module can be removed from the running kernel by using kldunload <module_name>.

To load the module during boot place the following into /boot/loader.conf:

```
nic_uio_load="YES"
```

Note: nic_uio_load="YES" must appear after the contigmem_load directive, if it exists.

By default, the nic_uio module will take ownership of network ports if they are recognized DPDK devices and are not owned by another module. However, since the FreeBSD kernel includes support, either built-in, or via a separate driver module, for most network card devices, it is likely that the ports to be used are already bound to a driver other than nic_uio. The following sub-section describe how to query and modify the device ownership of the ports to be used by DPDK applications.

Binding Network Ports to the nic_uio Module

Device ownership can be viewed using the pciconf -l command. The example below shows four Intel® 82599 network ports under if_ixgbe module ownership.

```
pciconf -l
ix0@pci0:1:0:0: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
ix1@pci0:1:0:1: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
ix2@pci0:2:0:0: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
ix3@pci0:2:0:1: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
```

The first column constitutes three components:

1. Device name: ixN

2. Unit name: `pci0`
3. Selector (Bus:Device:Function): `1:0:0`

Where no driver is associated with a device, the device name will be `none`.

By default, the FreeBSD kernel will include built-in drivers for the most common devices; a kernel rebuild would normally be required to either remove the drivers or configure them as loadable modules.

To avoid building a custom kernel, the `nic_uio` module can detach a network port from its current device driver. This is achieved by setting the `hw.nic_uio.bdfs` kernel environment variable prior to loading `nic_uio`, as follows:

```
kenv hw.nic_uio.bdfs="b:d:f,b:d:f,..."
```

Where a comma separated list of selectors is set, the list must not contain any whitespace.

For example to re-bind `ix2@pci0:2:0:0` and `ix3@pci0:2:0:1` to the `nic_uio` module upon loading, use the following command:

```
kenv hw.nic_uio.bdfs="2:0:0,2:0:1"
```

The variable can also be specified during boot by placing the following into `/boot/loader.conf`, before the previously-described `nic_uio_load` line - as shown:

```
hw.nic_uio.bdfs="2:0:0,2:0:1"
nic_uio_load="YES"
```

Binding Network Ports Back to their Original Kernel Driver

If the original driver for a network port has been compiled into the kernel, it is necessary to reboot FreeBSD to restore the original device binding. Before doing so, update or remove the `hw.nic_uio.bdfs` in `/boot/loader.conf`.

If rebinding to a driver that is a loadable module, the network port binding can be reset without rebooting. To do so, unload both the target kernel module and the `nic_uio` module, modify or clear the `hw.nic_uio.bdfs` kernel environment (kenv) value, and reload the two drivers - first the original kernel driver, and then the `nic_uio driver`. Note: the latter does not need to be reloaded unless there are ports that are still to be bound to it.

Example commands to perform these steps are shown below:

```
kldunload nic_uio
kldunload <original_driver>

# To clear the value completely:
kenv -u hw.nic_uio.bdfs

# To update the list of ports to bind:
kenv hw.nic_uio.bdfs="b:d:f,b:d:f,..."

kldload <original_driver>

kldload nic_uio  # optional
```

2.4 Compiling and Running Sample Applications

The chapter describes how to compile and run applications in a DPDK environment. It also provides a pointer to where sample applications are stored.

2.4.1 Compiling a Sample Application

The DPDK example applications make use of the pkg-config file installed on the system when DPDK is installed, and so can be built using GNU make.

Note: BSD make cannot be used to compile the DPDK example applications. GNU make can be installed using *pkg install gmake* if not already installed on the FreeBSD system.

The following shows how to compile the helloworld example app, following the installation of DPDK using *ninja install* as described previously:

```
$ export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig

$ cd examples/helloworld/

$ gmake
cc -O3 -I/usr/local/include -include rte_config.h -march=native
-D__BSD_VISIBLE  main.c -o build/helloworld-shared
-L/usr/local/lib -lrte_telemetry -lrte_bpf -lrte_flow_classify
-lrte_pipeline -lrte_table -lrte_port -lrte_fib -lrte_ipsec
-lrte_stack -lrte_security -lrte_sched -lrte_reorder -lrte_rib
-lrte_rcu -lrte_rawdev -lrte_pdump -lrte_member -lrte_lpm
-lrte_latencystats -lrte_jobstats -lrte_ip_frag -lrte_gso -lrte_gro
-lrte_eventdev -lrte_efd -lrte_distributor -lrte_cryptodev
-lrte_compressdev -lrte_cfgfile -lrte_bitratestats -lrte_bbdev
-lrte_acl -lrte_timer -lrte_hash -lrte_metrics -lrte_cmdline
-lrte_pci -lrte_ethdev -lrte_meter -lrte_net -lrte_mbuf
-lrte_mempool -lrte_ring -lrte_eal -lrte_kvargs
ln -sf helloworld-shared build/helloworld
```

2.4.2 Running a Sample Application

1. The `contigmem` and `nic_uio` modules must be set up prior to running an application.
2. Any ports to be used by the application must be already bound to the `nic_uio` module, as described in section *Binding Network Ports to the nic_uio Module*, prior to running the application. The application is linked with the DPDK target environment's Environment Abstraction Layer (EAL) library, which provides some options that are generic to every DPDK application.

A large number of options can be given to the EAL when running an application. A full list of options can be got by passing *–help* to a DPDK application. Some of the EAL options for FreeBSD are as follows:

- `-c COREMASK` or `-l CORELIST`: A hexadecimal bit mask of the cores to run on. Note that core numbering can change between platforms and should be determined beforehand. The corelist is a list of cores to use instead of a core mask.
- `-b <domain:bus:devid.func>`: Blacklisting of ports; prevent EAL from using specified PCI device (multiple `-b` options are allowed).

- `--use-device`: Use the specified Ethernet device(s) only. Use comma-separate `[domain:]bus:devid.func` values. Cannot be used with `-b` option.
- `-v`: Display version information on startup.
- `-m MB`: Memory to allocate from hugepages, regardless of processor socket.

Other options, specific to Linux and are not supported under FreeBSD are as follows:

- `socket-mem`: Memory to allocate from hugepages on specific sockets.
- `--huge-dir`: The directory where hugetlbfs is mounted.
- `mbuf-pool-ops-name`: Pool ops name for mbuf to use.
- `--file-prefix`: The prefix text used for hugepage filenames.

The `-c` or `-l` option is mandatory; the others are optional.

2.4.3 Running DPDK Applications Without Root Privileges

Although applications using the DPDK use network ports and other hardware resources directly, with a number of small permission adjustments, it is possible to run these applications as a user other than "root". To do so, the ownership, or permissions, on the following file system objects should be adjusted to ensure that the user account being used to run the DPDK application has access to them:

- The userspace-io device files in `/dev`, for example, `/dev/uio0`, `/dev/uio1`, and so on
- The userspace contiguous memory device: `/dev/contigmem`

Note: Please refer to the DPDK Release Notes for supported applications.

2.5 EAL parameters

This document contains a list of all EAL parameters. These parameters can be used by any DPDK application running on FreeBSD.

2.5.1 Common EAL parameters

The following EAL parameters are common to all platforms supported by DPDK.

Lcore-related options

- `-c <core mask>`

  Set the hexadecimal bitmask of the cores to run on.

- `-l <core list>`

  List of cores to run on

  The argument format is `<c1>[-c2][,c3[-c4],...]` where `c1`, `c2`, etc are core indexes between 0 and 128.

- `--lcores <core map>`

  Map lcore set to physical cpu set

  The argument format is:

  ```
  <lcores[@cpus]>[<,lcores[@cpus]>...]
  ```

  Lcore and CPU lists are grouped by `(` and `)` Within the group. The `-` character is used as a range separator and `,` is used as a single number separator. The grouping `()` can be omitted for single element group. The `@` can be omitted if cpus and lcores have the same value.

Note: At a given instance only one core option `--lcores`, `-l` or `-c` can be used.

- `--master-lcore <core ID>`

  Core ID that is used as master.

- `-s <service core mask>`

  Hexadecimal bitmask of cores to be used as service cores.

Device-related options

- `-b, --pci-blacklist <[domain:]bus:devid.func>`

  Blacklist a PCI device to prevent EAL from using it. Multiple -b options are allowed.

Note: PCI blacklist cannot be used with `-w` option.

- `-w, --pci-whitelist <[domain:]bus:devid.func>`

  Add a PCI device in white list.

Note: PCI whitelist cannot be used with `-b` option.

- `--vdev <device arguments>`

  Add a virtual device using the format:

  ```
  <driver><id>[,key=val, ...]
  ```

  For example:

  ```
  --vdev 'net_pcap0,rx_pcap=input.pcap,tx_pcap=output.pcap'
  ```

- `-d <path to shared object or directory>`

  Load external drivers. An argument can be a single shared object file, or a directory containing multiple driver shared objects. Multiple -d options are allowed.

- `--no-pci`

  Disable PCI bus.

Multiprocessing-related options

- `--proc-type <primary|secondary|auto>`

  Set the type of the current process.

- `--base-virtaddr <address>`

  Attempt to use a different starting address for all memory maps of the primary DPDK process. This can be helpful if secondary processes cannot start due to conflicts in address map.

Memory-related options

- `-n <number of channels>`

  Set the number of memory channels to use.

- `-r <number of ranks>`

  Set the number of memory ranks (auto-detected by default).

- `-m <megabytes>`

  Amount of memory to preallocate at startup.

- `--in-memory`

  Do not create any shared data structures and run entirely in memory. Implies `--no-shconf` and (if applicable) `--huge-unlink`.

- `--iova-mode <pa|va>`

  Force IOVA mode to a specific value.

Debugging options

- `--no-shconf`

  No shared files created (implies no secondary process support).

- `--no-huge`

  Use anonymous memory instead of hugepages (implies no secondary process support).

- `--log-level <type:val>`

  Specify log level for a specific component. For example:

```
--log-level lib.eal:debug
```

  Can be specified multiple times.

- `--trace=<regex-match>`

  Enable trace based on regular expression trace name. By default, the trace is disabled. User must specify this option to enable trace. For example:

  Global trace configuration for EAL only:

```
--trace=eal
```

Global trace configuration for ALL the components:

```
--trace=.*
```

Can be specified multiple times up to 32 times.

- `--trace-dir=<directory path>`

  Specify trace directory for trace output. For example:

  Configuring `/tmp/` as a trace output directory:

  ```
  --trace-dir=/tmp
  ```

  By default, trace output will created at `home` directory and parameter must be specified once only.

- `--trace-bufsz=<val>`

  Specify maximum size of allocated memory for trace output for each thread. Valid unit can be either `B` or `K` or `M` for `Bytes`, `KBytes` and `MBytes` respectively. For example:

  Configuring `2MB` as a maximum size for trace output file:

  ```
  --trace-bufsz=2M
  ```

  By default, size of trace output file is `1MB` and parameter must be specified once only.

- `--trace-mode=<o[verwrite] | d[iscard] >`

  Specify the mode of update of trace output file. Either update on a file can be wrapped or discarded when file size reaches its maximum limit. For example:

  To `discard` update on trace output file:

  ```
  --trace-mode=d or --trace-mode=discard
  ```

  Default mode is `overwrite` and parameter must be specified once only.

Other options

- `-h`, `--help`

  Display help message listing all EAL parameters.

- `-v`

  Display the version information on startup.

- `mbuf-pool-ops-name`:

  Pool ops name for mbuf to use.

- `--telemetry`:

  Enable telemetry (enabled by default).

- `--no-telemetry`:

  Disable telemetry.

2.5.2 FreeBSD-specific EAL parameters

There are currently no FreeBSD-specific EAL command-line parameters available.

CHAPTER

THREE

GETTING STARTED GUIDE FOR WINDOWS

3.1 Introduction

This document contains instructions for installing and configuring the Data Plane Development Kit (DPDK) software. The document describes how to compile and run a DPDK application in a Windows* OS application environment, without going deeply into detail.

*Other names and brands may be claimed as the property of others.

3.2 Limitations

DPDK for Windows is currently a work in progress. Not all DPDK source files compile. Support is being added in pieces so as to limit the overall scope of any individual patch series. The goal is to be able to run any DPDK application natively on Windows.

3.3 Compiling the DPDK Target from Source

3.3.1 System Requirements

Building the DPDK and its applications requires one of the following environments:

- The Clang-LLVM C compiler and Microsoft MSVC linker.
- The MinGW-w64 toolchain (either native or cross).

The Meson Build system is used to prepare the sources for compilation with the Ninja backend. The installation of these tools is covered in this section.

3.3.2 Option 1. Clang-LLVM C Compiler and Microsoft MSVC Linker

Install the Compiler

Download and install the clang compiler from LLVM website. For example, Clang-LLVM direct download link:

```
http://releases.llvm.org/7.0.1/LLVM-7.0.1-win64.exe
```

Install the Linker

Download and install the Build Tools for Visual Studio to link and build the files on windows, from Microsoft website. When installing build tools, select the "Visual C++ build tools" option and ensure the Windows SDK is selected.

3.3.3 Option 2. MinGW-w64 Toolchain

Obtain the latest version from MinGW-w64 website. On Windows, install to a folder without spaces in its name, like `C:\MinGW`. This path is assumed for the rest of this guide.

Version 4.0.4 for Ubuntu 16.04 cannot be used due to a MinGW-w64 bug.

3.3.4 Install the Build System

Download and install the build system from Meson website. A good option to choose is the MSI installer for both meson and ninja together:

```
http://mesonbuild.com/Getting-meson.html#installing-meson-and-ninja-with-the-msi-installer%22
```

Recommended version is either Meson 0.47.1 (baseline) or the latest release.

3.3.5 Install the Backend

If using Ninja, download and install the backend from Ninja website or install along with the meson build system.

3.3.6 Build the code

The build environment is setup to build the EAL and the helloworld example by default.

Option 1. Native Build on Windows

When using Clang-LLVM, specifying the compiler might be required to complete the meson command:

```
set CC=clang
```

When using MinGW-w64, it is sufficient to have toolchain executables in PATH:

```
set PATH=C:\MinGW\mingw64\bin;%PATH%
```

To compile the examples, the flag `-Dexamples` is required.

```
cd C:\Users\me\dpdk
meson -Dexamples=helloworld build
ninja -C build
```

Option 2. Cross-Compile with MinGW-w64

The cross-file option must be specified for Meson. Depending on the distribution, paths in this file may need adjustments.

```
meson --cross-file config/x86/meson_mingw.txt -Dexamples=helloworld build
ninja -C build
```

3.4 Run the helloworld example

Navigate to the examples in the build directory and run *dpdk-helloworld.exe*.

```
cd C:\Users\me\dpdk\build\examples
dpdk-helloworld.exe
hello from core 1
hello from core 3
hello from core 0
hello from core 2
```

Note for MinGW-w64: applications are linked to `libwinpthread-1.dll` by default. To run the example, either add toolchain executables directory to the PATH or copy the library to the working directory. Alternatively, static linking may be used (mind the LGPLv2.1 license).

CHAPTER

FOUR

SAMPLE APPLICATIONS USER GUIDES

4.1 Introduction to the DPDK Sample Applications

The DPDK Sample Applications are small standalone applications which demonstrate various features of DPDK. They can be considered as a cookbook of DPDK features. Users interested in getting started with DPDK can take the applications, try out the features, and then extend them to fit their needs.

4.1.1 Running Sample Applications

Some sample applications may have their own command-line parameters described in their respective guides, however all of them also share the same EAL parameters. Please refer to *EAL parameters (Linux)* or *EAL parameters (FreeBSD)* for a list of available EAL command-line options.

4.1.2 The DPDK Sample Applications

There are many sample applications available in the examples directory of DPDK. These examples range from simple to reasonably complex but most are designed to demonstrate one particular feature of DPDK. Some of the more interesting examples are highlighted below.

- *Hello World*: As with most introductions to a programming framework a good place to start is with the Hello World application. The Hello World example sets up the DPDK Environment Abstraction Layer (EAL), and prints a simple "Hello World" message to each of the DPDK enabled cores. This application doesn't do any packet forwarding but it is a good way to test if the DPDK environment is compiled and set up properly.
- *Basic Forwarding/Skeleton Application*: The Basic Forwarding/Skeleton contains the minimum amount of code required to enable basic packet forwarding with DPDK. This allows you to test if your network interfaces are working with DPDK.
- *Network Layer 2 forwarding*: The Network Layer 2 forwarding, or `l2fwd` application does forwarding based on Ethernet MAC addresses like a simple switch.
- *Network Layer 2 forwarding*: The Network Layer 2 forwarding, or `l2fwd-event` application does forwarding based on Ethernet MAC addresses like a simple switch. It demonstrates usage of poll and event mode IO mechanism under a single application.
- *Network Layer 3 forwarding*: The Network Layer3 forwarding, or `l3fwd` application does forwarding based on Internet Protocol, IPv4 or IPv6 like a simple router.
- *Network Layer 3 forwarding Graph*: The Network Layer3 forwarding Graph, or `l3fwd_graph` application does forwarding based on IPv4 like a simple router with DPDK Graph framework.

- *Hardware packet copying*: The Hardware packet copying, or `ioatfwd` application demonstrates how to use IOAT rawdev driver for copying packets between two threads.
- *Packet Distributor*: The Packet Distributor demonstrates how to distribute packets arriving on an Rx port to different cores for processing and transmission.
- *Multi-Process Application*: The multi-process application shows how two DPDK processes can work together using queues and memory pools to share information.
- *RX/TX callbacks Application*: The RX/TX callbacks sample application is a packet forwarding application that demonstrates the use of user defined callbacks on received and transmitted packets. The application calculates the latency of a packet between RX (packet arrival) and TX (packet transmission) by adding callbacks to the RX and TX packet processing functions.
- *IPsec Security Gateway*: The IPsec Security Gateway application is minimal example of something closer to a real world example. This is also a good example of an application using the DPDK Cryptodev framework.
- *Precision Time Protocol (PTP) client*: The PTP client is another minimal implementation of a real world application. In this case the application is a PTP client that communicates with a PTP master clock to synchronize time on a Network Interface Card (NIC) using the IEEE1588 protocol.
- *Quality of Service (QoS) Scheduler*: The QoS Scheduler application demonstrates the use of DPDK to provide QoS scheduling.

There are many more examples shown in the following chapters. Each of the documented sample applications show how to compile, configure and run the application as well as explaining the main functionality of the code.

4.2 Compiling the Sample Applications

This section explains how to compile the DPDK sample applications.

4.2.1 To compile all the sample applications

Set the path to DPDK source code if its not set:

```
export RTE_SDK=/path/to/rte_sdk
```

Go to DPDK source:

```
cd $RTE_SDK
```

Build DPDK:

```
make defconfig
make
```

Build the sample applications:

```
export RTE_TARGET=build
make -C examples
```

For other possible RTE_TARGET values and additional information on compiling see *Compiling DPDK on Linux* or *Compiling DPDK on FreeBSD*. Applications are output to: $RTE_SDK/examples/app-dir/build or $RTE_SDK/examples/app-dir/$RTE_TARGET.

In the example above the compiled application is written to the build subdirectory. To have the applications written to a different location, the O=/path/to/build/directory option may be specified in the make command.

```
make O=/tmp
```

To build the applications for debugging use the DEBUG option. This option adds some extra flags, disables compiler optimizations and sets verbose output.

```
make DEBUG=1
```

4.2.2 To compile a single application

Set the path to DPDK source code:

```
export RTE_SDK=/path/to/rte_sdk
```

Go to DPDK source:

```
cd $RTE_SDK
```

Build DPDK:

```
make defconfig
make
```

Go to the sample application directory. Unless otherwise specified the sample applications are located in $RTE_SDK/examples/.

Build the application:

```
export RTE_TARGET=build
make
```

4.2.3 To cross compile the sample application(s)

For cross compiling the sample application(s), please append 'CROSS=$(CROSS_COMPILER_PREFIX)' to the 'make' command. In example of AARCH64 cross compiling:

```
export RTE_TARGET=build
export RTE_SDK=/path/to/rte_sdk
make -C examples CROSS=aarch64-linux-gnu-
       or
make CROSS=aarch64-linux-gnu-
```

4.3 Command Line Sample Application

This chapter describes the Command Line sample application that is part of the Data Plane Development Kit (DPDK).

4.3.1 Overview

The Command Line sample application is a simple application that demonstrates the use of the command line interface in the DPDK. This application is a readline-like interface that can be used to debug a DPDK application, in a Linux* application environment.

Note: The rte_cmdline library should not be used in production code since it is not validated to the same standard as other DPDK libraries. See also the "rte_cmdline library should not be used in production code due to limited testing" item in the "Known Issues" section of the Release Notes.

The Command Line sample application supports some of the features of the GNU readline library such as, completion, cut/paste and some other special bindings that make configuration and debug faster and easier.

The application shows how the rte_cmdline application can be extended to handle a list of objects. There are three simple commands:

- add obj_name IP: Add a new object with an IP/IPv6 address associated to it.
- del obj_name: Delete the specified object.
- show obj_name: Show the IP associated with the specified object.

Note: To terminate the application, use **Ctrl-d**.

4.3.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*

The application is located in the `cmd_line` sub-directory.

4.3.3 Running the Application

To run the application in linux environment, issue the following command:

```
$ ./build/cmdline -l 0-3 -n 4
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.3.4 Explanation

The following sections provide some explanation of the code.

EAL Initialization and cmdline Start

The first task is the initialization of the Environment Abstraction Layer (EAL). This is achieved as follows:

```
int main(int argc, char **argv)
{
    ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_panic("Cannot init EAL\n");
```

Then, a new command line object is created and started to interact with the user through the console:

```
cl = cmdline_stdin_new(main_ctx, "example> ");
cmdline_interact(cl);
cmdline_stdin_exit(cl);
```

The cmd line_interact() function returns when the user types **Ctrl-d** and in this case, the application exits.

Defining a cmdline Context

A cmdline context is a list of commands that are listed in a NULL-terminated table, for example:

```
cmdline_parse_ctx_t main_ctx[] = {
    (cmdline_parse_inst_t *) &cmd_obj_del_show,
    (cmdline_parse_inst_t *) &cmd_obj_add,
    (cmdline_parse_inst_t *) &cmd_help,
     NULL,
};
```

Each command (of type cmdline_parse_inst_t) is defined statically. It contains a pointer to a callback function that is executed when the command is parsed, an opaque pointer, a help string and a list of tokens in a NULL-terminated table.

The rte_cmdline application provides a list of pre-defined token types:

- String Token: Match a static string, a list of static strings or any string.
- Number Token: Match a number that can be signed or unsigned, from 8-bit to 32-bit.
- IP Address Token: Match an IPv4 or IPv6 address or network.
- Ethernet* Address Token: Match a MAC address.

In this example, a new token type obj_list is defined and implemented in the parse_obj_list.c and parse_obj_list.h files.

For example, the cmd_obj_del_show command is defined as shown below:

```
struct cmd_obj_add_result {
    cmdline_fixed_string_t action;
    cmdline_fixed_string_t name;
    struct object *obj;
};
```

(continues on next page)

(continued from previous page)

```
static void cmd_obj_del_show_parsed(void *parsed_result, struct cmdline *cl, __rte_unused void␣
↪*data)
{
   /* ... */
}

cmdline_parse_token_string_t cmd_obj_action = TOKEN_STRING_INITIALIZER(struct cmd_obj_del_show_
↪result, action, "show#del");

parse_token_obj_list_t cmd_obj_obj = TOKEN_OBJ_LIST_INITIALIZER(struct cmd_obj_del_show_result,
↪ obj, &global_obj_list);

cmdline_parse_inst_t cmd_obj_del_show = {
    .f = cmd_obj_del_show_parsed, /* function to call */
    .data = NULL,  /* 2nd arg of func */
    .help_str = "Show/del an object",
    .tokens = { /* token list, NULL terminated */
        (void *)&cmd_obj_action,
        (void *)&cmd_obj_obj,
         NULL,
    },
};
```

This command is composed of two tokens:

- The first token is a string token that can be show or del.
- The second token is an object that was previously added using the add command in the global_obj_list variable.

Once the command is parsed, the rte_cmdline application fills a cmd_obj_del_show_result structure. A pointer to this structure is given as an argument to the callback function and can be used in the body of this function.

4.4 Ethtool Sample Application

The Ethtool sample application shows an implementation of an ethtool-like API and provides a console environment that allows its use to query and change Ethernet card parameters. The sample is based upon a simple L2 frame reflector.

4.4.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ethtool` sub-directory.

4.4.2 Running the Application

The application requires an available core for each port, plus one. The only available options are the standard ones for the EAL:

```
./ethtool-app/ethtool-app/${RTE_TARGET}/ethtool [EAL options]
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.4.3 Using the application

The application is console-driven using the cmdline DPDK interface:

```
EthApp>
```

From this interface the available commands and descriptions of what they do as follows:

- `drvinfo`: Print driver info
- `eeprom`: Dump EEPROM to file
- `module-eeprom`: Dump plugin module EEPROM to file
- `link`: Print port link states
- `macaddr`: Gets/sets MAC address
- `mtu`: Set NIC MTU
- `open`: Open port
- `pause`: Get/set port pause state
- `portstats`: Print port statistics
- `regs`: Dump port register(s) to file
- `ringparam`: Get/set ring parameters
- `rxmode`: Toggle port Rx mode
- `stop`: Stop port
- `validate`: Check that given MAC address is valid unicast address
- `vlan`: Add/remove VLAN id
- `quit`: Exit program

4.4.4 Explanation

The sample program has two parts: A background *packet reflector* that runs on a slave core, and a foreground *Ethtool Shell* that runs on the master core. These are described below.

Packet Reflector

The background packet reflector is intended to demonstrate basic packet processing on NIC ports controlled by the Ethtool shim. Each incoming MAC frame is rewritten so that it is returned to the sender, using the port in question's own MAC address as the source address, and is then sent out on the same port.

Ethtool Shell

The foreground part of the Ethtool sample is a console-based interface that accepts commands as described in *using the application*. Individual call-back functions handle the detail associated with each command, which make use of the functions defined in the *Ethtool interface* to the DPDK functions.

4.4.5 Ethtool interface

The Ethtool interface is built as a separate library, and implements the following functions:

- `rte_ethtool_get_drvinfo()`
- `rte_ethtool_get_regs_len()`
- `rte_ethtool_get_regs()`
- `rte_ethtool_get_link()`
- `rte_ethtool_get_eeprom_len()`
- `rte_ethtool_get_eeprom()`
- `rte_ethtool_set_eeprom()`
- `rte_ethtool_get_module_info()`
- `rte_ethtool_get_module_eeprom()`
- `rte_ethtool_get_pauseparam()`
- `rte_ethtool_set_pauseparam()`
- `rte_ethtool_net_open()`
- `rte_ethtool_net_stop()`
- `rte_ethtool_net_get_mac_addr()`
- `rte_ethtool_net_set_mac_addr()`
- `rte_ethtool_net_validate_addr()`
- `rte_ethtool_net_change_mtu()`
- `rte_ethtool_net_get_stats64()`
- `rte_ethtool_net_vlan_rx_add_vid()`

- rte_ethtool_net_vlan_rx_kill_vid()
- rte_ethtool_net_set_rx_mode()
- rte_ethtool_get_ringparam()
- rte_ethtool_set_ringparam()

4.5 Hello World Sample Application

The Hello World sample application is an example of the simplest DPDK application that can be written. The application simply prints an "helloworld" message on every enabled lcore.

4.5.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `helloworld` sub-directory.

4.5.2 Running the Application

To run the example in a linux environment:

```
$ ./build/helloworld -l 0-3 -n 4
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.5.3 Explanation

The following sections provide some explanation of code.

EAL Initialization

The first task is to initialize the Environment Abstraction Layer (EAL). This is done in the main() function using the following code:

```
int

main(int argc, char **argv)

{
    ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_panic("Cannot init EAL\n");
```

This call finishes the initialization process that was started before main() is called (in case of a Linux environment). The argc and argv arguments are provided to the rte_eal_init() function. The value returned is the number of parsed arguments.

Starting Application Unit Lcores

Once the EAL is initialized, the application is ready to launch a function on an lcore. In this example, lcore_hello() is called on every available lcore. The following is the definition of the function:

```
static int
lcore_hello(__rte_unused void *arg)
{
    unsigned lcore_id;

    lcore_id = rte_lcore_id();
    printf("hello from core %u\n", lcore_id);
    return 0;
}
```

The code that launches the function on each lcore is as follows:

```
/* call lcore_hello() on every slave lcore */

RTE_LCORE_FOREACH_SLAVE(lcore_id) {
   rte_eal_remote_launch(lcore_hello, NULL, lcore_id);
}

/* call it on master lcore too */

lcore_hello(NULL);
```

The following code is equivalent and simpler:

```
rte_eal_mp_remote_launch(lcore_hello, NULL, CALL_MASTER);
```

Refer to the *DPDK API Reference* for detailed information on the rte_eal_mp_remote_launch() function.

4.6 Basic Forwarding Sample Application

The Basic Forwarding sample application is a simple *skeleton* example of a forwarding application.

It is intended as a demonstration of the basic components of a DPDK forwarding application. For more detailed implementations see the L2 and L3 forwarding sample applications.

4.6.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `skeleton` sub-directory.

4.6.2 Running the Application

To run the example in a `linux` environment:

```
./build/basicfwd -l 1 -n 4
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.6.3 Explanation

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with `rte_` and are explained in detail in the *DPDK API Documentation*.

The Main Function

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

The `main()` also allocates a mempool to hold the mbufs (Message Buffers) used by the application:

```
mbuf_pool = rte_mempool_create("MBUF_POOL",
                               NUM_MBUFS * nb_ports,
                               MBUF_SIZE,
                               MBUF_CACHE_SIZE,
                               sizeof(struct rte_pktmbuf_pool_private),
                               rte_pktmbuf_pool_init, NULL,
                               rte_pktmbuf_init,      NULL,
                               rte_socket_id(),
                               0);
```

Mbufs are the packet buffer structure used by DPDK. They are explained in detail in the "Mbuf Library" section of the *DPDK Programmer's Guide*.

The `main()` function also initializes all the ports using the user defined `port_init()` function which is explained in the next section:

```
RTE_ETH_FOREACH_DEV(portid) {
    if (port_init(portid, mbuf_pool) != 0) {
        rte_exit(EXIT_FAILURE,
                 "Cannot init port %" PRIu8 "\n", portid);
    }
}
```

Once the initialization is complete, the application is ready to launch a function on an lcore. In this example `lcore_main()` is called on a single lcore.

```
lcore_main();
```

The `lcore_main()` function is explained below.

The Port Initialization Function

The main functional part of the port initialization used in the Basic Forwarding application is shown below:

```
static inline int
port_init(uint16_t port, struct rte_mempool *mbuf_pool)
{
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;
    struct rte_ether_addr addr;
    int retval;
    uint16_t q;

    if (!rte_eth_dev_is_valid_port(port))
        return -1;

    /* Configure the Ethernet device. */
    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
    if (retval != 0)
        return retval;

    /* Allocate and set up 1 RX queue per Ethernet port. */
    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL, mbuf_pool);
        if (retval < 0)
            return retval;
    }

    /* Allocate and set up 1 TX queue per Ethernet port. */
    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL);
        if (retval < 0)
            return retval;
    }

    /* Start the Ethernet port. */
    retval = rte_eth_dev_start(port);
    if (retval < 0)
        return retval;

    /* Enable RX in promiscuous mode for the Ethernet device. */
    retval = rte_eth_promiscuous_enable(port);
    if (retval != 0)
        return retval;

    return 0;
}
```

The Ethernet ports are configured with default settings using the `rte_eth_dev_configure()` function and the `port_conf_default` struct:

```
static const struct rte_eth_conf port_conf_default = {
    .rxmode = { .max_rx_pkt_len = RTE_ETHER_MAX_LEN }
};
```

For this example the ports are set up with 1 RX and 1 TX queue using the `rte_eth_rx_queue_setup()` and `rte_eth_tx_queue_setup()` functions.

The Ethernet port is then started:

```
retval  = rte_eth_dev_start(port);
```

Finally the RX port is set in promiscuous mode:

```
retval = rte_eth_promiscuous_enable(port);
```

The Lcores Main

As we saw above the `main()` function calls an application function on the available lcores. For the Basic Forwarding application the lcore function looks like the following:

```
static __rte_noreturn void
lcore_main(void)
{
    uint16_t port;

    /*
     * Check that the port is on the same NUMA node as the polling thread
     * for best performance.
     */
    RTE_ETH_FOREACH_DEV(port)
        if (rte_eth_dev_socket_id(port) > 0 &&
                rte_eth_dev_socket_id(port) !=
                        (int)rte_socket_id())
            printf("WARNING, port %u is on remote NUMA node to "
                    "polling thread.\n\tPerformance will "
                    "not be optimal.\n", port);

    printf("\nCore %u forwarding packets. [Ctrl+C to quit]\n",
            rte_lcore_id());

    /* Run until the application is quit or killed. */
    for (;;) {
        /*
         * Receive packets on a port and forward them on the paired
         * port. The mapping is 0 -> 1, 1 -> 0, 2 -> 3, 3 -> 2, etc.
         */
        RTE_ETH_FOREACH_DEV(port) {

            /* Get burst of RX packets, from first port of pair. */
            struct rte_mbuf *bufs[BURST_SIZE];
            const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                    bufs, BURST_SIZE);

            if (unlikely(nb_rx == 0))
                continue;

            /* Send burst of TX packets, to second port of pair. */
            const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
```

(continues on next page)

(continued from previous page)

```
                    bufs, nb_rx);

            /* Free any unsent packets. */
            if (unlikely(nb_tx < nb_rx)) {
                uint16_t buf;
                for (buf = nb_tx; buf < nb_rx; buf++)
                    rte_pktmbuf_free(bufs[buf]);
            }
        }
    }
}
```

The main work of the application is done within the loop:

```
for (;;) {
    RTE_ETH_FOREACH_DEV(port) {

        /* Get burst of RX packets, from first port of pair. */
        struct rte_mbuf *bufs[BURST_SIZE];
        const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                bufs, BURST_SIZE);

        if (unlikely(nb_rx == 0))
            continue;

        /* Send burst of TX packets, to second port of pair. */
        const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                bufs, nb_rx);

        /* Free any unsent packets. */
        if (unlikely(nb_tx < nb_rx)) {
            uint16_t buf;
            for (buf = nb_tx; buf < nb_rx; buf++)
                rte_pktmbuf_free(bufs[buf]);
        }
    }
}
```

Packets are received in bursts on the RX ports and transmitted in bursts on the TX ports. The ports are grouped in pairs with a simple mapping scheme using the an XOR on the port number:

```
0 -> 1
1 -> 0

2 -> 3
3 -> 2

etc.
```

The `rte_eth_tx_burst()` function frees the memory buffers of packets that are transmitted. If packets fail to transmit, `(nb_tx < nb_rx)`, then they must be freed explicitly using `rte_pktmbuf_free()`.

The forwarding loop can be interrupted and the application closed using `Ctrl-C`.

4.7 RX/TX Callbacks Sample Application

The RX/TX Callbacks sample application is a packet forwarding application that demonstrates the use of user defined callbacks on received and transmitted packets. The application performs a simple latency check, using callbacks, to determine the time packets spend within the application.

In the sample application a user defined callback is applied to all received packets to add a timestamp. A separate callback is applied to all packets prior to transmission to calculate the elapsed time, in CPU cycles.

If hardware timestamping is supported by the NIC, the sample application will also display the average latency since the packet was timestamped in hardware, on top of the latency since the packet was received and processed by the RX callback.

4.7.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `rxtx_callbacks` sub-directory.

The callbacks feature requires that the `CONFIG_RTE_ETHDEV_RXTX_CALLBACKS` setting is on in the `config/common_` config file that applies to the target. This is generally on by default:

```
CONFIG_RTE_ETHDEV_RXTX_CALLBACKS=y
```

4.7.2 Running the Application

To run the example in a `linux` environment:

```
./build/rxtx_callbacks -l 1 -n 4 -- [-t]
```

Use -t to enable hardware timestamping. If not supported by the NIC, an error will be displayed.

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.7.3 Explanation

The `rxtx_callbacks` application is mainly a simple forwarding application based on the *Basic Forwarding Sample Application*. See that section of the documentation for more details of the forwarding part of the application.

The sections below explain the additional RX/TX callback code.

The Main Function

The `main()` function performs the application initialization and calls the execution threads for each lcore. This function is effectively identical to the `main()` function explained in *Basic Forwarding Sample Application*.

The `lcore_main()` function is also identical.

The main difference is in the user defined `port_init()` function where the callbacks are added. This is explained in the next section:

The Port Initialization Function

The main functional part of the port initialization is shown below with comments:

```
static inline int
port_init(uint16_t port, struct rte_mempool *mbuf_pool)
{
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;
    struct rte_ether_addr addr;
    int retval;
    uint16_t q;

    /* Configure the Ethernet device. */
    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
    if (retval != 0)
        return retval;

    /* Allocate and set up 1 RX queue per Ethernet port. */
    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL, mbuf_pool);
        if (retval < 0)
            return retval;
    }

    /* Allocate and set up 1 TX queue per Ethernet port. */
    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL);
        if (retval < 0)
            return retval;
    }

    /* Start the Ethernet port. */
    retval = rte_eth_dev_start(port);
    if (retval < 0)
        return retval;

    /* Enable RX in promiscuous mode for the Ethernet device. */
    retval = rte_eth_promiscuous_enable(port);
    if (retval != 0)
        return retval;

    /* Add the callbacks for RX and TX.*/
    rte_eth_add_rx_callback(port, 0, add_timestamps, NULL);
    rte_eth_add_tx_callback(port, 0, calc_latency, NULL);

```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

The RX and TX callbacks are added to the ports/queues as function pointers:

```
rte_eth_add_rx_callback(port, 0, add_timestamps, NULL);
rte_eth_add_tx_callback(port, 0, calc_latency,   NULL);
```

More than one callback can be added and additional information can be passed to callback function pointers as a `void*`. In the examples above `NULL` is used.

The `add_timestamps()` and `calc_latency()` functions are explained below.

The add_timestamps() Callback

The `add_timestamps()` callback is added to the RX port and is applied to all packets received:

```
static uint16_t
add_timestamps(uint16_t port __rte_unused, uint16_t qidx __rte_unused,
        struct rte_mbuf **pkts, uint16_t nb_pkts, void *_ __rte_unused)
{
    unsigned i;
    uint64_t now = rte_rdtsc();

    for (i = 0; i < nb_pkts; i++)
        pkts[i]->udata64 = now;

    return nb_pkts;
}
```

The DPDK function `rte_rdtsc()` is used to add a cycle count timestamp to each packet (see the *cycles* section of the *DPDK API Documentation* for details).

The calc_latency() Callback

The `calc_latency()` callback is added to the TX port and is applied to all packets prior to transmission:

```
static uint16_t
calc_latency(uint16_t port __rte_unused, uint16_t qidx __rte_unused,
        struct rte_mbuf **pkts, uint16_t nb_pkts, void *_ __rte_unused)
{
    uint64_t cycles = 0;
    uint64_t now = rte_rdtsc();
    unsigned i;

    for (i = 0; i < nb_pkts; i++)
        cycles += now - pkts[i]->udata64;

    latency_numbers.total_cycles += cycles;
    latency_numbers.total_pkts   += nb_pkts;

    if (latency_numbers.total_pkts > (100 * 1000 * 1000ULL)) {
        printf("Latency = %"PRIu64" cycles\n",
                latency_numbers.total_cycles / latency_numbers.total_pkts);

        latency_numbers.total_cycles = latency_numbers.total_pkts = 0;
```

(continues on next page)

(continued from previous page)

```
    }

    return nb_pkts;
}
```

The `calc_latency()` function accumulates the total number of packets and the total number of cycles used. Once more than 100 million packets have been transmitted the average cycle count per packet is printed out and the counters are reset.

4.8 Flow Classify Sample Application

The Flow Classify sample application is based on the simple *skeleton* example of a forwarding application.

It is intended as a demonstration of the basic components of a DPDK forwarding application which uses the Flow Classify library API's.

Please refer to the *Flow Classification Library* for more information.

4.8.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `flow_classify` sub-directory.

4.8.2 Running the Application

To run the example in a `linux` environment:

```
cd ~/dpdk/examples/flow_classify
./build/flow_classify -c 4 -n 4 -- --rule_ipv4="../ipv4_rules_file.txt"
```

Please refer to the *DPDK Getting Started Guide*, section *Compiling and Running Sample Applications* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.8.3 Sample ipv4_rules_file.txt

```
#file format:
#src_ip/masklen dst_ip/masklen src_port : mask dst_port : mask proto/mask priority
#
2.2.2.3/24 2.2.2.7/24 32 : 0xffff 33 : 0xffff 17/0xff 0
9.9.9.3/24 9.9.9.7/24 32 : 0xffff 33 : 0xffff 17/0xff 1
9.9.9.3/24 9.9.9.7/24 32 : 0xffff 33 : 0xffff 6/0xff 2
9.9.8.3/24 9.9.8.7/24 32 : 0xffff 33 : 0xffff 6/0xff 3
6.7.8.9/24 2.3.4.5/24 32 : 0x0000 33 : 0x0000 132/0xff 4
```

4.8.4 Explanation

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with `rte_` and are explained in detail in the *DPDK API Documentation*.

ACL field definitions for the IPv4 5 tuple rule

The following field definitions are used when creating the ACL table during initialisation of the `Flow Classify` application..

```
 enum {
     PROTO_FIELD_IPV4,
     SRC_FIELD_IPV4,
     DST_FIELD_IPV4,
     SRCP_FIELD_IPV4,
     DSTP_FIELD_IPV4,
     NUM_FIELDS_IPV4
};

enum {
    PROTO_INPUT_IPV4,
    SRC_INPUT_IPV4,
    DST_INPUT_IPV4,
    SRCP_DESTP_INPUT_IPV4
};

static struct rte_acl_field_def ipv4_defs[NUM_FIELDS_IPV4] = {
    /* first input field - always one byte long. */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof(uint8_t),
        .field_index = PROTO_FIELD_IPV4,
        .input_index = PROTO_INPUT_IPV4,
        .offset = sizeof(struct rte_ether_hdr) +
            offsetof(struct rte_ipv4_hdr, next_proto_id),
    },
    /* next input field (IPv4 source address) - 4 consecutive bytes. */
    {
        /* rte_flow uses a bit mask for IPv4 addresses */
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof(uint32_t),
        .field_index = SRC_FIELD_IPV4,
        .input_index = SRC_INPUT_IPV4,
        .offset = sizeof(struct rte_ether_hdr) +
            offsetof(struct rte_ipv4_hdr, src_addr),
    },
    /* next input field (IPv4 destination address) - 4 consecutive bytes. */
    {
        /* rte_flow uses a bit mask for IPv4 addresses */
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof(uint32_t),
        .field_index = DST_FIELD_IPV4,
        .input_index = DST_INPUT_IPV4,
        .offset = sizeof(struct rte_ether_hdr) +
            offsetof(struct rte_ipv4_hdr, dst_addr),
    },
    /*
     * Next 2 fields (src & dst ports) form 4 consecutive bytes.
```

(continues on next page)

(continued from previous page)

```
     * They share the same input index.
     */
    {
        /* rte_flow uses a bit mask for protocol ports */
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof(uint16_t),
        .field_index = SRCP_FIELD_IPV4,
        .input_index = SRCP_DESTP_INPUT_IPV4,
        .offset = sizeof(struct rte_ether_hdr) +
            sizeof(struct rte_ipv4_hdr) +
            offsetof(struct rte_tcp_hdr, src_port),
    },
    {
         /* rte_flow uses a bit mask for protocol ports */
         .type = RTE_ACL_FIELD_TYPE_BITMASK,
         .size = sizeof(uint16_t),
         .field_index = DSTP_FIELD_IPV4,
         .input_index = SRCP_DESTP_INPUT_IPV4,
         .offset = sizeof(struct rte_ether_hdr) +
             sizeof(struct rte_ipv4_hdr) +
             offsetof(struct rte_tcp_hdr, dst_port),
    },
};
```

The Main Function

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

It then parses the flow_classify application arguments

```
ret = parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid flow_classify parameters\n");
```

The `main()` function also allocates a mempool to hold the mbufs (Message Buffers) used by the application:

```
mbuf_pool = rte_mempool_create("MBUF_POOL",
                               NUM_MBUFS * nb_ports,
                               MBUF_SIZE,
                               MBUF_CACHE_SIZE,
                               sizeof(struct rte_pktmbuf_pool_private),
                               rte_pktmbuf_pool_init, NULL,
                               rte_pktmbuf_init, NULL,
                               rte_socket_id(),
                               0);
```

mbufs are the packet buffer structure used by DPDK. They are explained in detail in the "Mbuf Library" section of the *DPDK Programmer's Guide*.

The `main()` function also initializes all the ports using the user defined `port_init()` function which is explained in the next section:

```
RTE_ETH_FOREACH_DEV(portid) {
    if (port_init(portid, mbuf_pool) != 0) {
        rte_exit(EXIT_FAILURE,
                 "Cannot init port %" PRIu8 "\n", portid);
    }
}
```

The `main()` function creates the `flow classifier object` and adds an `ACL table` to the flow classifier.

```
struct flow_classifier {
    struct rte_flow_classifier *cls;
};

struct flow_classifier_acl {
    struct flow_classifier cls;
} __rte_cache_aligned;

/* Memory allocation */
size = RTE_CACHE_LINE_ROUNDUP(sizeof(struct flow_classifier_acl));
cls_app = rte_zmalloc(NULL, size, RTE_CACHE_LINE_SIZE);
if (cls_app == NULL)
    rte_exit(EXIT_FAILURE, "Cannot allocate classifier memory\n");

cls_params.name = "flow_classifier";
cls_params.socket_id = socket_id;

cls_app->cls = rte_flow_classifier_create(&cls_params);
if (cls_app->cls == NULL) {
    rte_free(cls_app);
    rte_exit(EXIT_FAILURE, "Cannot create classifier\n");
}

/* initialise ACL table params */
table_acl_params.name = "table_acl_ipv4_5tuple";
table_acl_params.n_rule_fields = RTE_DIM(ipv4_defs);
table_acl_params.n_rules = FLOW_CLASSIFY_MAX_RULE_NUM;
memcpy(table_acl_params.field_format, ipv4_defs, sizeof(ipv4_defs));

/* initialise table create params */
cls_table_params.ops = &rte_table_acl_ops,
cls_table_params.arg_create = &table_acl_params,
cls_table_params.type = RTE_FLOW_CLASSIFY_TABLE_ACL_IP4_5TUPLE;

ret = rte_flow_classify_table_create(cls_app->cls, &cls_table_params);
if (ret) {
    rte_flow_classifier_free(cls_app->cls);
    rte_free(cls);
    rte_exit(EXIT_FAILURE, "Failed to create classifier table\n");
}
```

It then reads the ipv4_rules_file.txt file and initialises the parameters for the `rte_flow_classify_table_entry_add` API. This API adds a rule to the ACL table.

```
if (add_rules(parm_config.rule_ipv4_name)) {
    rte_flow_classifier_free(cls_app->cls);
    rte_free(cls_app);
    rte_exit(EXIT_FAILURE, "Failed to add rules\n");
```

(continues on next page)

(continued from previous page)

```
}
```

Once the initialization is complete, the application is ready to launch a function on an lcore. In this example `lcore_main()` is called on a single lcore.

```
lcore_main(cls_app);
```

The `lcore_main()` function is explained below.

The Port Initialization Function

The main functional part of the port initialization used in the Basic Forwarding application is shown below:

```
static inline int
port_init(uint8_t port, struct rte_mempool *mbuf_pool)
{
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;
    struct rte_ether_addr addr;
    int retval;
    uint16_t q;

    /* Configure the Ethernet device. */
    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
    if (retval != 0)
        return retval;

    /* Allocate and set up 1 RX queue per Ethernet port. */
    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL, mbuf_pool);
        if (retval < 0)
            return retval;
    }

    /* Allocate and set up 1 TX queue per Ethernet port. */
    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL);
        if (retval < 0)
            return retval;
    }

    /* Start the Ethernet port. */
    retval = rte_eth_dev_start(port);
    if (retval < 0)
        return retval;

    /* Display the port MAC address. */
    retval = rte_eth_macaddr_get(port, &addr);
    if (retval < 0)
        return retval;
    printf("Port %u MAC: %02" PRIx8 " %02" PRIx8 " %02" PRIx8
           " %02" PRIx8 " %02" PRIx8 " %02" PRIx8 "\n",
           port,
           addr.addr_bytes[0], addr.addr_bytes[1],
           addr.addr_bytes[2], addr.addr_bytes[3],
```

(continues on next page)

(continued from previous page)

```
            addr.addr_bytes[4], addr.addr_bytes[5]);

    /* Enable RX in promiscuous mode for the Ethernet device. */
    retval = rte_eth_promiscuous_enable(port);
    if (retval != 0)
            return retval;

    return 0;
}
```

The Ethernet ports are configured with default settings using the `rte_eth_dev_configure()` function and the `port_conf_default` struct.

```
static const struct rte_eth_conf port_conf_default = {
    .rxmode = { .max_rx_pkt_len = RTE_ETHER_MAX_LEN }
};
```

For this example the ports are set up with 1 RX and 1 TX queue using the `rte_eth_rx_queue_setup()` and `rte_eth_tx_queue_setup()` functions.

The Ethernet port is then started:

```
retval  = rte_eth_dev_start(port);
```

Finally the RX port is set in promiscuous mode:

```
retval = rte_eth_promiscuous_enable(port);
```

The Add Rules function

The `add_rules` function reads the `ipv4_rules_file.txt` file and calls the `add_classify_rule` function which calls the `rte_flow_classify_table_entry_add` API.

```
static int
add_rules(const char *rule_path)
{
    FILE *fh;
    char buff[LINE_MAX];
    unsigned int i = 0;
    unsigned int total_num = 0;
    struct rte_eth_ntuple_filter ntuple_filter;

    fh = fopen(rule_path, "rb");
    if (fh == NULL)
        rte_exit(EXIT_FAILURE, "%s: Open %s failed\n", __func__,
                 rule_path);

    fseek(fh, 0, SEEK_SET);

    i = 0;
    while (fgets(buff, LINE_MAX, fh) != NULL) {
        i++;

        if (is_bypass_line(buff))
            continue;

        if (total_num >= FLOW_CLASSIFY_MAX_RULE_NUM - 1) {
```

(continues on next page)

(continued from previous page)

```
            printf("\nINFO: classify rule capacity %d reached\n",
                   total_num);
            break;
        }

        if (parse_ipv4_5tuple_rule(buff, &ntuple_filter) != 0)
            rte_exit(EXIT_FAILURE,
                     "%s Line %u: parse rules error\n",
                     rule_path, i);

        if (add_classify_rule(&ntuple_filter) != 0)
            rte_exit(EXIT_FAILURE, "add rule error\n");

        total_num++;
    }

    fclose(fh);
    return 0;
}
```

The Lcore Main function

As we saw above the `main()` function calls an application function on the available lcores. The `lcore_main` function calls the `rte_flow_classifier_query` API. For the Basic Forwarding application the `lcore_main` function looks like the following:

```
/* flow classify data */
static int num_classify_rules;
static struct rte_flow_classify_rule *rules[MAX_NUM_CLASSIFY];
static struct rte_flow_classify_ipv4_5tuple_stats ntuple_stats;
static struct rte_flow_classify_stats classify_stats = {
        .stats = (void *)&ntuple_stats
};

static __rte_noreturn void
lcore_main(cls_app)
{
    uint16_t port;

    /*
     * Check that the port is on the same NUMA node as the polling thread
     * for best performance.
     */
    RTE_ETH_FOREACH_DEV(port)
        if (rte_eth_dev_socket_id(port) > 0 &&
            rte_eth_dev_socket_id(port) != (int)rte_socket_id()) {
            printf("\n\n");
            printf("WARNING: port %u is on remote NUMA node\n",
                   port);
            printf("to polling thread.\n");
            printf("Performance will not be optimal.\n");

            printf("\nCore %u forwarding packets. \n",
                   rte_lcore_id());
            printf("[Ctrl+C to quit]\n
        }

    /* Run until the application is quit or killed. */
```

(continues on next page)

(continued from previous page)

```
    for (;;) {
        /*
         * Receive packets on a port and forward them on the paired
         * port. The mapping is 0 -> 1, 1 -> 0, 2 -> 3, 3 -> 2, etc.
         */
        RTE_ETH_FOREACH_DEV(port) {

            /* Get burst of RX packets, from first port of pair. */
            struct rte_mbuf *bufs[BURST_SIZE];
            const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                    bufs, BURST_SIZE);

            if (unlikely(nb_rx == 0))
                continue;

            for (i = 0; i < MAX_NUM_CLASSIFY; i++) {
                if (rules[i]) {
                    ret = rte_flow_classifier_query(
                        cls_app->cls,
                        bufs, nb_rx, rules[i],
                        &classify_stats);
                    if (ret)
                        printf(
                            "rule [%d] query failed ret [%d]\n\n",
                            i, ret);
                    else {
                        printf(
                            "rule[%d] count=%"PRIu64"\n",
                            i, ntuple_stats.counter1);

                        printf("proto = %d\n",
                            ntuple_stats.ipv4_5tuple.proto);
                    }
                 }
             }

            /* Send burst of TX packets, to second port of pair. */
            const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                    bufs, nb_rx);

            /* Free any unsent packets. */
            if (unlikely(nb_tx < nb_rx)) {
                uint16_t buf;
                for (buf = nb_tx; buf < nb_rx; buf++)
                    rte_pktmbuf_free(bufs[buf]);
            }
        }
    }
}
```

The main work of the application is done within the loop:

```
for (;;) {
    RTE_ETH_FOREACH_DEV(port) {

        /* Get burst of RX packets, from first port of pair. */
        struct rte_mbuf *bufs[BURST_SIZE];
        const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                bufs, BURST_SIZE);
```

(continues on next page)

(continued from previous page)

```
        if (unlikely(nb_rx == 0))
            continue;

        /* Send burst of TX packets, to second port of pair. */
        const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                bufs, nb_rx);

        /* Free any unsent packets. */
        if (unlikely(nb_tx < nb_rx)) {
            uint16_t buf;
            for (buf = nb_tx; buf < nb_rx; buf++)
                rte_pktmbuf_free(bufs[buf]);
        }
    }
}
```

Packets are received in bursts on the RX ports and transmitted in bursts on the TX ports. The ports are grouped in pairs with a simple mapping scheme using the an XOR on the port number:

```
0 -> 1
1 -> 0

2 -> 3
3 -> 2

etc.
```

The `rte_eth_tx_burst()` function frees the memory buffers of packets that are transmitted. If packets fail to transmit, `(nb_tx < nb_rx)`, then they must be freed explicitly using `rte_pktmbuf_free()`.

The forwarding loop can be interrupted and the application closed using `Ctrl-C`.

4.9 Basic RTE Flow Filtering Sample Application

The Basic RTE flow filtering sample application is a simple example of a creating a RTE flow rule.

It is intended as a demonstration of the basic components RTE flow rules.

4.9.1 Compiling the Application

To compile the application export the path to the DPDK source tree and go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk

cd ${RTE_SDK}/examples/flow_filtering
```

Set the target, for example:

```
export RTE_TARGET=x86_64-native-linux-gcc
```

See the *DPDK Getting Started* Guide for possible `RTE_TARGET` values.

Build the application as follows:

```
make
```

4.9.2 Running the Application

To run the example in a `linux` environment:

```
./build/flow -l 1 -n 1
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.9.3 Explanation

The example is built from 2 files, `main.c` which holds the example logic and `flow_blocks.c` that holds the implementation for building the flow rule.

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with `rte_` and are explained in detail in the *DPDK API Documentation*.

The Main Function

The `main()` function located in `main.c` file performs the initialization and runs the main loop function.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

The `main()` also allocates a mempool to hold the mbufs (Message Buffers) used by the application:

```
mbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", 4096, 128, 0,
                                    RTE_MBUF_DEFAULT_BUF_SIZE,
                                    rte_socket_id());
```

Mbufs are the packet buffer structure used by DPDK. They are explained in detail in the "Mbuf Library" section of the *DPDK Programmer's Guide*.

The `main()` function also initializes all the ports using the user defined `init_port()` function which is explained in the next section:

```
init_port();
```

Once the initialization is complete, we set the flow rule using the following code:

```
/* create flow for send packet with */
flow = generate_ipv4_flow(port_id, selected_queue,
                          SRC_IP, EMPTY_MASK,
                          DEST_IP, FULL_MASK, &error);
if (!flow) {
```

(continues on next page)

(continued from previous page)

```
        printf("Flow can't be created %d message: %s\n",
                        error.type,
                        error.message ? error.message : "(no stated reason)");
        rte_exit(EXIT_FAILURE, "error in creating flow");
}
```

In the last part the application is ready to launch the `main_loop()` function. Which is explained below.

```
main_loop();
```

The Port Initialization Function

The main functional part of the port initialization used in the flow filtering application is shown below:

```
init_port(void)
{
        int ret;
        uint16_t i;
        struct rte_eth_conf port_conf = {
                .rxmode = {
                        .split_hdr_size = 0,
                        },
                .txmode = {
                        .offloads =
                                DEV_TX_OFFLOAD_VLAN_INSERT |
                                DEV_TX_OFFLOAD_IPV4_CKSUM  |
                                DEV_TX_OFFLOAD_UDP_CKSUM   |
                                DEV_TX_OFFLOAD_TCP_CKSUM   |
                                DEV_TX_OFFLOAD_SCTP_CKSUM  |
                                DEV_TX_OFFLOAD_TCP_TSO,
                },
        };
        struct rte_eth_txconf txq_conf;
        struct rte_eth_rxconf rxq_conf;
        struct rte_eth_dev_info dev_info;

        printf(":: initializing port: %d\n", port_id);
        ret = rte_eth_dev_configure(port_id,
                nr_queues, nr_queues, &port_conf);
        if (ret < 0) {
                rte_exit(EXIT_FAILURE,
                        ":: cannot configure device: err=%d, port=%u\n",
                        ret, port_id);
        }

        rte_eth_dev_info_get(port_id, &dev_info);
        rxq_conf = dev_info.default_rxconf;
        rxq_conf.offloads = port_conf.rxmode.offloads;
        /* only set Rx queues: something we care only so far */
        for (i = 0; i < nr_queues; i++) {
                ret = rte_eth_rx_queue_setup(port_id, i, 512,
                        rte_eth_dev_socket_id(port_id),
                        &rxq_conf,
                        mbuf_pool);
                if (ret < 0) {
                         rte_exit(EXIT_FAILURE,
                                ":: Rx queue setup failed: err=%d, port=%u\n",
                                ret, port_id);
```

(continues on next page)

(continued from previous page)

```
                }
        }

        txq_conf = dev_info.default_txconf;
        txq_conf.offloads = port_conf.txmode.offloads;

        for (i = 0; i < nr_queues; i++) {
                ret = rte_eth_tx_queue_setup(port_id, i, 512,
                        rte_eth_dev_socket_id(port_id),
                        &txq_conf);
                if (ret < 0) {
                        rte_exit(EXIT_FAILURE,
                                ":: Tx queue setup failed: err=%d, port=%u\n",
                                ret, port_id);
                }
       }

        ret = rte_eth_promiscuous_enable(port_id);
        if (ret != 0) {
                rte_exit(EXIT_FAILURE,
                        ":: cannot enable promiscuous mode: err=%d, port=%u\n",
                        ret, port_id);
        }

        ret = rte_eth_dev_start(port_id);
        if (ret < 0) {
                rte_exit(EXIT_FAILURE,
                        "rte_eth_dev_start:err=%d, port=%u\n",
                        ret, port_id);
        }

        assert_link_status();

        printf(":: initializing port: %d done\n", port_id);
}
```

The Ethernet port is configured with default settings using the `rte_eth_dev_configure()` function and the `port_conf_default` struct:

```
struct rte_eth_conf port_conf = {
        .rxmode = {
                .split_hdr_size = 0,
                },
        .txmode = {
                .offloads =
                        DEV_TX_OFFLOAD_VLAN_INSERT |
                        DEV_TX_OFFLOAD_IPV4_CKSUM  |
                        DEV_TX_OFFLOAD_UDP_CKSUM   |
                        DEV_TX_OFFLOAD_TCP_CKSUM   |
                        DEV_TX_OFFLOAD_SCTP_CKSUM  |
                        DEV_TX_OFFLOAD_TCP_TSO,
                },
        };

ret = rte_eth_dev_configure(port_id, nr_queues, nr_queues, &port_conf);
if (ret < 0) {
     rte_exit(EXIT_FAILURE,
               ":: cannot configure device: err=%d, port=%u\n",
               ret, port_id);
}
```

(continues on next page)

(continued from previous page)

```
rte_eth_dev_info_get(port_id, &dev_info);
rxq_conf = dev_info.default_rxconf;
rxq_conf.offloads = port_conf.rxmode.offloads;
```

For this example we are configuring number of rx and tx queues that are connected to a single port.

```
for (i = 0; i < nr_queues; i++) {
       ret = rte_eth_rx_queue_setup(port_id, i, 512,
                                    rte_eth_dev_socket_id(port_id),
                                    &rxq_conf,
                                    mbuf_pool);
       if (ret < 0) {
               rte_exit(EXIT_FAILURE,
                       ":: Rx queue setup failed: err=%d, port=%u\n",
                       ret, port_id);
       }
}

for (i = 0; i < nr_queues; i++) {
       ret = rte_eth_tx_queue_setup(port_id, i, 512,
                                    rte_eth_dev_socket_id(port_id),
                                    &txq_conf);
       if (ret < 0) {
               rte_exit(EXIT_FAILURE,
                        ":: Tx queue setup failed: err=%d, port=%u\n",
                        ret, port_id);
       }
}
```

In the next step we create and apply the flow rule. which is to send packets with destination ip equals to 192.168.1.1 to queue number 1. The detail explanation of the `generate_ipv4_flow()` appears later in this document:

```
flow = generate_ipv4_flow(port_id, selected_queue,
                          SRC_IP, EMPTY_MASK,
                          DEST_IP, FULL_MASK, &error);
```

We are setting the RX port to promiscuous mode:

```
ret = rte_eth_promiscuous_enable(port_id);
if (ret != 0) {
     rte_exit(EXIT_FAILURE,
              ":: cannot enable promiscuous mode: err=%d, port=%u\n",
              ret, port_id);
}
```

The last step is to start the port.

```
ret = rte_eth_dev_start(port_id);
if (ret < 0)  {
     rte_exit(EXIT_FAILURE, "rte_eth_dev_start:err%d, port=%u\n",
                     ret, port_id);
}
```

The main_loop function

As we saw above the `main()` function calls an application function to handle the main loop. For the flow filtering application the main_loop function looks like the following:

```
static void
main_loop(void)
{
        struct rte_mbuf *mbufs[32];
        struct rte_ether_hdr *eth_hdr;
        uint16_t nb_rx;
        uint16_t i;
        uint16_t j;

        while (!force_quit) {
                for (i = 0; i < nr_queues; i++) {
                        nb_rx = rte_eth_rx_burst(port_id,
                                                 i, mbufs, 32);
                        if (nb_rx) {
                                for (j = 0; j < nb_rx; j++) {
                                        struct rte_mbuf *m = mbufs[j];

                                        eth_hdr = rte_pktmbuf_mtod(m,
                                                     struct rte_ether_hdr *);
                                        print_ether_addr("src=",
                                                     &eth_hdr->s_addr);
                                        print_ether_addr(" - dst=",
                                                     &eth_hdr->d_addr);
                                        printf(" - queue=0x%x",
                                                         (unsigned int)i);
                                        printf("\n");
                                        rte_pktmbuf_free(m);
                                }
                        }
                }
        }
        /* closing and releasing resources */
        rte_flow_flush(port_id, &error);
        rte_eth_dev_stop(port_id);
        rte_eth_dev_close(port_id);
}
```

The main work of the application is reading the packets from all queues and printing for each packet the destination queue:

```
while (!force_quit) {
    for (i = 0; i < nr_queues; i++) {
                nb_rx = rte_eth_rx_burst(port_id, i, mbufs, 32);
            if (nb_rx) {
                    for (j = 0; j < nb_rx; j++) {
                         struct rte_mbuf *m = mbufs[j];
                         eth_hdr = rte_pktmbuf_mtod(m, struct rte_ether_hdr *);
                         print_ether_addr("src=", &eth_hdr->s_addr);
                         print_ether_addr(" - dst=", &eth_hdr->d_addr);
                         printf(" - queue=0x%x", (unsigned int)i);
                         printf("\n");
                         rte_pktmbuf_free(m);
                    }
            }
       }
}
```

The forwarding loop can be interrupted and the application closed using `Ctrl-C`. Which results in closing the port and the device using `rte_eth_dev_stop` and `rte_eth_dev_close`

The generate_ipv4_flow function

The generate_ipv4_flow function is responsible for creating the flow rule. This function is located in the `flow_blocks.c` file.

```
static struct rte_flow *
generate_ipv4_flow(uint8_t port_id, uint16_t rx_q,
                uint32_t src_ip, uint32_t src_mask,
                uint32_t dest_ip, uint32_t dest_mask,
                struct rte_flow_error *error)
{
        struct rte_flow_attr attr;
        struct rte_flow_item pattern[MAX_PATTERN_NUM];
        struct rte_flow_action action[MAX_ACTION_NUM];
        struct rte_flow *flow = NULL;
        struct rte_flow_action_queue queue = { .index = rx_q };
        struct rte_flow_item_ipv4 ip_spec;
        struct rte_flow_item_ipv4 ip_mask;

        memset(pattern, 0, sizeof(pattern));
        memset(action, 0, sizeof(action));

        /*
         * set the rule attribute.
         * in this case only ingress packets will be checked.
         */
        memset(&attr, 0, sizeof(struct rte_flow_attr));
        attr.ingress = 1;

        /*
         * create the action sequence.
         * one action only,  move packet to queue
         */
        action[0].type = RTE_FLOW_ACTION_TYPE_QUEUE;
        action[0].conf = &queue;
        action[1].type = RTE_FLOW_ACTION_TYPE_END;

        /*
         * set the first level of the pattern (ETH).
         * since in this example we just want to get the
         * ipv4 we set this level to allow all.
         */
        pattern[0].type = RTE_FLOW_ITEM_TYPE_ETH;

        /*
         * setting the second level of the pattern (IP).
         * in this example this is the level we care about
         * so we set it according to the parameters.
         */
        memset(&ip_spec, 0, sizeof(struct rte_flow_item_ipv4));
        memset(&ip_mask, 0, sizeof(struct rte_flow_item_ipv4));
        ip_spec.hdr.dst_addr = htonl(dest_ip);
        ip_mask.hdr.dst_addr = dest_mask;
        ip_spec.hdr.src_addr = htonl(src_ip);
        ip_mask.hdr.src_addr = src_mask;
        pattern[1].type = RTE_FLOW_ITEM_TYPE_IPV4;
        pattern[1].spec = &ip_spec;
```

(continues on next page)

(continued from previous page)

```
        pattern[1].mask = &ip_mask;

        /* the final level must be always type end */
        pattern[2].type = RTE_FLOW_ITEM_TYPE_END;

        int res = rte_flow_validate(port_id, &attr, pattern, action, error);
        if(!res)
            flow = rte_flow_create(port_id, &attr, pattern, action, error);

        return flow;
}
```

The first part of the function is declaring the structures that will be used.

```
struct rte_flow_attr attr;
struct rte_flow_item pattern[MAX_PATTERN_NUM];
struct rte_flow_action action[MAX_ACTION_NUM];
struct rte_flow *flow;
struct rte_flow_error error;
struct rte_flow_action_queue queue = { .index = rx_q };
struct rte_flow_item_ipv4 ip_spec;
struct rte_flow_item_ipv4 ip_mask;
```

The following part create the flow attributes, in our case ingress.

```
memset(&attr, 0, sizeof(struct rte_flow_attr));
attr.ingress = 1;
```

The third part defines the action to be taken when a packet matches the rule. In this case send the packet to queue.

```
action[0].type = RTE_FLOW_ACTION_TYPE_QUEUE;
action[0].conf = &queue;
action[1].type = RTE_FLOW_ACTION_TYPE_END;
```

The fourth part is responsible for creating the pattern and is built from number of steps. In each step we build one level of the pattern starting with the lowest one.

Setting the first level of the pattern ETH:

```
pattern[0].type = RTE_FLOW_ITEM_TYPE_ETH;
```

Setting the second level of the pattern IP:

```
memset(&ip_spec, 0, sizeof(struct rte_flow_item_ipv4));
memset(&ip_mask, 0, sizeof(struct rte_flow_item_ipv4));
ip_spec.hdr.dst_addr = htonl(dest_ip);
ip_mask.hdr.dst_addr = dest_mask;
ip_spec.hdr.src_addr = htonl(src_ip);
ip_mask.hdr.src_addr = src_mask;
pattern[1].type = RTE_FLOW_ITEM_TYPE_IPV4;
pattern[1].spec = &ip_spec;
pattern[1].mask = &ip_mask;
```

Closing the pattern part.

```
pattern[2].type = RTE_FLOW_ITEM_TYPE_END;
```

The last part of the function is to validate the rule and create it.

```
int res = rte_flow_validate(port_id, &attr, pattern, action, &error);
if (!res)
     flow = rte_flow_create(port_id, &attr, pattern, action, &error);
```

4.10 IP Fragmentation Sample Application

The IPv4 Fragmentation application is a simple example of packet processing using the Data Plane Development Kit (DPDK). The application does L3 forwarding with IPv4 and IPv6 packet fragmentation.

4.10.1 Overview

The application demonstrates the use of zero-copy buffers for packet fragmentation. The initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. This guide highlights the differences between the two applications.

There are three key differences from the L2 Forwarding sample application:

- The first difference is that the IP Fragmentation sample application makes use of indirect buffers.
- The second difference is that the forwarding decision is taken based on information read from the input packet's IP header.
- The third difference is that the application differentiates between IP and non-IP traffic by means of offload flags.

The Longest Prefix Match (LPM for IPv4, LPM6 for IPv6) table is used to store/lookup an outgoing port number, associated with that IP address. Any unmatched packets are forwarded to the originating port.

By default, input frame sizes up to 9.5 KB are supported. Before forwarding, the input IP packet is fragmented to fit into the "standard" Ethernet* v2 MTU (1500 bytes).

4.10.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ip_fragmentation` sub-directory.

4.10.3 Running the Application

The LPM object is created and loaded with the pre-configured entries read from global l3fwd_ipv4_route_array and l3fwd_ipv6_route_array tables. For each input packet, the packet forwarding decision (that is, the identification of the output interface for the packet) is taken as a result of LPM lookup. If the IP packet size is greater than default output MTU, then the input packet is fragmented and several fragments are sent via the output interface.

Application usage:

```
./build/ip_fragmentation [EAL options] -- -p PORTMASK [-q NQ]
```

where:

- -p PORTMASK is a hexadecimal bitmask of ports to configure

- -q NQ is the number of queue (=ports) per lcore (the default is 1)

To run the example in linux environment with 2 lcores (2,4) over 2 ports(0,2) with 1 RX queue per lcore:

```
./build/ip_fragmentation -l 2,4 -n 3 -- -p 5
EAL: coremask set to 14
EAL: Detected lcore 0 on socket 0
EAL: Detected lcore 1 on socket 1
EAL: Detected lcore 2 on socket 0
EAL: Detected lcore 3 on socket 1
EAL: Detected lcore 4 on socket 0
...

Initializing port 0 on lcore 2... Address:00:1B:21:76:FA:2C, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 1
Initializing port 2 on lcore 4... Address:00:1B:21:5C:FF:54, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 3IP_FRAG: Socket 0: adding route 100.10.0.0/16 (port 0)
IP_FRAG: Socket 0: adding route 100.20.0.0/16 (port 1)
...
IP_FRAG: Socket 0: adding route 0101:0101:0101:0101:0101:0101:0101:0101/48 (port 0)
IP_FRAG: Socket 0: adding route 0201:0101:0101:0101:0101:0101:0101:0101/48 (port 1)
...
IP_FRAG: entering main loop on lcore 4
IP_FRAG: -- lcoreid=4 portid=2
IP_FRAG: entering main loop on lcore 2
IP_FRAG: -- lcoreid=2 portid=0
```

To run the example in linux environment with 1 lcore (4) over 2 ports(0,2) with 2 RX queues per lcore:

```
./build/ip_fragmentation -l 4 -n 3 -- -p 5 -q 2
```

To test the application, flows should be set up in the flow generator that match the values in the l3fwd_ipv4_route_array and/or l3fwd_ipv6_route_array table.

The default l3fwd_ipv4_route_array table is:

```
struct l3fwd_ipv4_route l3fwd_ipv4_route_array[] = {
    {RTE_IPV4(100, 10, 0, 0), 16, 0},
    {RTE_IPV4(100, 20, 0, 0), 16, 1},
    {RTE_IPV4(100, 30, 0, 0), 16, 2},
    {RTE_IPV4(100, 40, 0, 0), 16, 3},
    {RTE_IPV4(100, 50, 0, 0), 16, 4},
    {RTE_IPV4(100, 60, 0, 0), 16, 5},
    {RTE_IPV4(100, 70, 0, 0), 16, 6},
    {RTE_IPV4(100, 80, 0, 0), 16, 7},
};
```

The default l3fwd_ipv6_route_array table is:

```
struct l3fwd_ipv6_route l3fwd_ipv6_route_array[] = {
    {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 0},
    {{2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 1},
    {{3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 2},
    {{4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 3},
    {{5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 4},
    {{6, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 5},
    {{7, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 6},
    {{8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 7},
};
```

For example, for the input IPv4 packet with destination address: 100.10.1.1 and packet length 9198 bytes, seven IPv4 packets will be sent out from port #0 to the destination address 100.10.1.1: six of those packets will have length 1500 bytes and one packet will have length 318 bytes. IP Fragmentation sample application provides basic NUMA support in that all the memory structures are allocated on all sockets that have active lcores on them.

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.11 IPv4 Multicast Sample Application

The IPv4 Multicast application is a simple example of packet processing using the Data Plane Development Kit (DPDK). The application performs L3 multicasting.

4.11.1 Overview

The application demonstrates the use of zero-copy buffers for packet forwarding. The initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. This guide highlights the differences between the two applications. There are two key differences from the L2 Forwarding sample application:

- The IPv4 Multicast sample application makes use of indirect buffers.
- The forwarding decision is taken based on information read from the input packet's IPv4 header.

The lookup method is the Four-byte Key (FBK) hash-based method. The lookup table is composed of pairs of destination IPv4 address (the FBK) and a port mask associated with that IPv4 address.

Note: The max port mask supported in the given hash table is 0xf, so only first four ports can be supported. If using non-consecutive ports, use the destination IPv4 address accordingly.

For convenience and simplicity, this sample application does not take IANA-assigned multicast addresses into account, but instead equates the last four bytes of the multicast group (that is, the last four bytes of the destination IP address) with the mask of ports to multicast packets to. Also, the application does not consider the Ethernet addresses; it looks only at the IPv4 destination address for any given packet.

4.11.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ipv4_multicast` sub-directory.

4.11.3 Running the Application

The application has a number of command line options:

```
./build/ipv4_multicast [EAL options] -- -p PORTMASK [-q NQ]
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -q NQ: determines the number of queues per lcore

Note: Unlike the basic L2/L3 Forwarding sample applications, NUMA support is not provided in the IPv4 Multicast sample application.

Typically, to run the IPv4 Multicast sample application, issue the following command (as root):

```
./build/ipv4_multicast -l 0-3 -n 3 -- -p 0x3 -q 1
```

In this command:

- The -l option enables cores 0, 1, 2 and 3
- The -n option specifies 3 memory channels
- The -p option enables ports 0 and 1
- The -q option assigns 1 queue to each lcore

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.11.4 Explanation

The following sections provide some explanation of the code. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. The following sections describe aspects that are specific to the IPv4 Multicast sample application.

Memory Pool Initialization

The IPv4 Multicast sample application uses three memory pools. Two of the pools are for indirect buffers used for packet duplication purposes. Memory pools for indirect buffers are initialized differently from the memory pool for direct buffers:

```
packet_pool = rte_pktmbuf_pool_create("packet_pool", NB_PKT_MBUF, 32,
                    0, PKT_MBUF_DATA_SIZE, rte_socket_id());
header_pool = rte_pktmbuf_pool_create("header_pool", NB_HDR_MBUF, 32,
                    0, HDR_MBUF_DATA_SIZE, rte_socket_id());
clone_pool = rte_pktmbuf_pool_create("clone_pool", NB_CLONE_MBUF, 32,
                    0, 0, rte_socket_id());
```

The reason for this is because indirect buffers are not supposed to hold any packet data and therefore can be initialized with lower amount of reserved memory for each buffer.

Hash Initialization

The hash object is created and loaded with the pre-configured entries read from a global array:

```
static int

init_mcast_hash(void)
{
    uint32_t i;
    mcast_hash_params.socket_id = rte_socket_id();

    mcast_hash = rte_fbk_hash_create(&mcast_hash_params);
    if (mcast_hash == NULL){
        return -1;
    }

    for (i = 0; i < N_MCAST_GROUPS; i ++){
        if (rte_fbk_hash_add_key(mcast_hash, mcast_group_table[i].ip, mcast_group_table[i].
↪port_mask) < 0) {
                    return -1;
        }
    }
    return 0;
}
```

Forwarding

All forwarding is done inside the mcast_forward() function. Firstly, the Ethernet* header is removed from the packet and the IPv4 address is extracted from the IPv4 header:

```
/* Remove the Ethernet header from the input packet */

iphdr = (struct rte_ipv4_hdr *)rte_pktmbuf_adj(m, sizeof(struct rte_ether_hdr));
RTE_ASSERT(iphdr != NULL);
dest_addr = rte_be_to_cpu_32(iphdr->dst_addr);
```

Then, the packet is checked to see if it has a multicast destination address and if the routing table has any ports assigned to the destination address:

```
if (!RTE_IS_IPV4_MCAST(dest_addr) ||
   (hash = rte_fbk_hash_lookup(mcast_hash, dest_addr)) <= 0 ||
   (port_mask = hash & enabled_port_mask) == 0) {
       rte_pktmbuf_free(m);
       return;
}
```

Then, the number of ports in the destination portmask is calculated with the help of the bitcnt() function:

```
/* Get number of bits set. */

static inline uint32_t bitcnt(uint32_t v)
{
    uint32_t n;

    for (n = 0; v != 0; v &= v - 1, n++)
        ;
    return n;
}
```

This is done to determine which forwarding algorithm to use. This is explained in more detail in the next section.

Thereafter, a destination Ethernet address is constructed:

```
/* construct destination Ethernet address */

dst_eth_addr = ETHER_ADDR_FOR_IPV4_MCAST(dest_addr);
```

Since Ethernet addresses are also part of the multicast process, each outgoing packet carries the same destination Ethernet address. The destination Ethernet address is constructed from the lower 23 bits of the multicast group OR-ed with the Ethernet address 01:00:5e:00:00:00, as per RFC 1112:

```
#define ETHER_ADDR_FOR_IPV4_MCAST(x) \
    (rte_cpu_to_be_64(0x01005e000000ULL | ((x) & 0x7fffff)) >> 16)
```

Then, packets are dispatched to the destination ports according to the portmask associated with a multicast group:

```
for (port = 0; use_clone != port_mask; port_mask >>= 1, port++) {
    /* Prepare output packet and send it out. */

    if ((port_mask & 1) != 0) {
        if (likely ((mc = mcast_out_pkt(m, use_clone)) != NULL))
            mcast_send_pkt(mc, &dst_eth_addr.as_addr, qconf, port);
        else if (use_clone == 0)
             rte_pktmbuf_free(m);
   }
}
```

The actual packet transmission is done in the mcast_send_pkt() function:

```
static inline void mcast_send_pkt(struct rte_mbuf *pkt, struct rte_ether_addr *dest_addr,␣
↪struct lcore_queue_conf *qconf, uint16_t port)
{
    struct rte_ether_hdr *ethdr;
    uint16_t len;

    /* Construct Ethernet header. */

    ethdr = (struct rte_ether_hdr *)rte_pktmbuf_prepend(pkt, (uint16_t) sizeof(*ethdr));

    RTE_ASSERT(ethdr != NULL);

    rte_ether_addr_copy(dest_addr, &ethdr->d_addr);
    rte_ether_addr_copy(&ports_eth_addr[port], &ethdr->s_addr);
    ethdr->ether_type = rte_be_to_cpu_16(RTE_ETHER_TYPE_IPV4);

    /* Put new packet into the output queue */

    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = pkt;
    qconf->tx_mbufs[port].len = ++len;

    /* Transmit packets */

    if (unlikely(MAX_PKT_BURST == len))
        send_burst(qconf, port);
}
```

Buffer Cloning

This is the most important part of the application since it demonstrates the use of zero- copy buffer cloning. There are two approaches for creating the outgoing packet and although both are based on the data zero-copy idea, there are some differences in the detail.

The first approach creates a clone of the input packet, for example, walk though all segments of the input packet and for each of segment, create a new buffer and attach that new buffer to the segment (refer to rte_pktmbuf_clone() in the rte_mbuf library for more details). A new buffer is then allocated for the packet header and is prepended to the cloned buffer.

The second approach does not make a clone, it just increments the reference counter for all input packet segment, allocates a new buffer for the packet header and prepends it to the input packet.

Basically, the first approach reuses only the input packet's data, but creates its own copy of packet's metadata. The second approach reuses both input packet's data and metadata.

The advantage of first approach is that each outgoing packet has its own copy of the metadata, so we can safely modify the data pointer of the input packet. That allows us to skip creation if the output packet is for the last destination port and instead modify input packet's header in place. For example, for N destination ports, we need to invoke mcast_out_pkt() (N-1) times.

The advantage of the second approach is that there is less work to be done for each outgoing packet, that is, the "clone" operation is skipped completely. However, there is a price to pay. The input packet's metadata must remain intact, so for N destination ports, we need to invoke mcast_out_pkt() (N) times.

Therefore, for a small number of outgoing ports (and segments in the input packet), first approach is faster. As the number of outgoing ports (and/or input segments) grows, the second approach becomes more preferable.

Depending on the number of segments or the number of ports in the outgoing portmask, either the first (with cloning) or the second (without cloning) approach is taken:

```
use_clone = (port_num <= MCAST_CLONE_PORTS && m->pkt.nb_segs <= MCAST_CLONE_SEGS);
```

It is the mcast_out_pkt() function that performs the packet duplication (either with or without actually cloning the buffers):

```
static inline struct rte_mbuf *mcast_out_pkt(struct rte_mbuf *pkt, int use_clone)
{
    struct rte_mbuf *hdr;

    /* Create new mbuf for the header. */

    if (unlikely ((hdr = rte_pktmbuf_alloc(header_pool)) == NULL))
        return NULL;

    /* If requested, then make a new clone packet. */

    if (use_clone != 0 && unlikely ((pkt = rte_pktmbuf_clone(pkt, clone_pool)) == NULL)) {
        rte_pktmbuf_free(hdr);
        return NULL;
    }

    /* prepend new header */

    hdr->pkt.next = pkt;

    /* update header's fields */
```

(continues on next page)

(continued from previous page)

```
    hdr->pkt.pkt_len = (uint16_t)(hdr->pkt.data_len + pkt->pkt.pkt_len);
    hdr->pkt.nb_segs = pkt->pkt.nb_segs + 1;

    /* copy metadata from source packet */

    hdr->pkt.in_port = pkt->pkt.in_port;
    hdr->pkt.vlan_macip = pkt->pkt.vlan_macip;
    hdr->pkt.hash = pkt->pkt.hash;
    rte_mbuf_sanity_check(hdr, RTE_MBUF_PKT, 1);

    return hdr;
}
```

4.12 IP Reassembly Sample Application

The L3 Forwarding application is a simple example of packet processing using the DPDK. The application performs L3 forwarding with reassembly for fragmented IPv4 and IPv6 packets.

4.12.1 Overview

The application demonstrates the use of the DPDK libraries to implement packet forwarding with reassembly for IPv4 and IPv6 fragmented packets. The initialization and run- time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. The main difference from the L2 Forwarding sample application is that it reassembles fragmented IPv4 and IPv6 packets before forwarding. The maximum allowed size of reassembled packet is 9.5 KB.

There are two key differences from the L2 Forwarding sample application:

- The first difference is that the forwarding decision is taken based on information read from the input packet's IP header.
- The second difference is that the application differentiates between IP and non-IP traffic by means of offload flags.

The Longest Prefix Match (LPM for IPv4, LPM6 for IPv6) table is used to store/lookup an outgoing port number, associated with that IPv4 address. Any unmatched packets are forwarded to the originating port.

4.12.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ip_reassembly` sub-directory.

4.12.3 Running the Application

The application has a number of command line options:

```
./build/ip_reassembly [EAL options] -- -p PORTMASK [-q NQ] [--maxflows=FLOWS>] [--
↪flowttl=TTL[(s|ms)]]
```

where:

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -q NQ: Number of RX queues per lcore
- –maxflows=FLOWS: determines maximum number of active fragmented flows (1-65535). Default value: 4096.
- –flowttl=TTL[(s|ms)]: determines maximum Time To Live for fragmented packet. If all fragments of the packet wouldn't appear within given time-out, then they are considered as invalid and will be dropped. Valid range is 1ms - 3600s. Default value: 1s.

To run the example in linux environment with 2 lcores (2,4) over 2 ports(0,2) with 1 RX queue per lcore:

```
./build/ip_reassembly -l 2,4 -n 3 -- -p 5
EAL: coremask set to 14
EAL: Detected lcore 0 on socket 0
EAL: Detected lcore 1 on socket 1
EAL: Detected lcore 2 on socket 0
EAL: Detected lcore 3 on socket 1
EAL: Detected lcore 4 on socket 0
...

Initializing port 0 on lcore 2... Address:00:1B:21:76:FA:2C, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 1
Initializing port 2 on lcore 4... Address:00:1B:21:5C:FF:54, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 3IP_FRAG: Socket 0: adding route 100.10.0.0/16 (port 0)
IP_RSMBL: Socket 0: adding route 100.20.0.0/16 (port 1)
...

IP_RSMBL: Socket 0: adding route 0101:0101:0101:0101:0101:0101:0101:0101/48 (port 0)
IP_RSMBL: Socket 0: adding route 0201:0101:0101:0101:0101:0101:0101:0101/48 (port 1)
...

IP_RSMBL: entering main loop on lcore 4
IP_RSMBL: -- lcoreid=4 portid=2
IP_RSMBL: entering main loop on lcore 2
IP_RSMBL: -- lcoreid=2 portid=0
```

To run the example in linux environment with 1 lcore (4) over 2 ports(0,2) with 2 RX queues per lcore:

```
./build/ip_reassembly -l 4 -n 3 -- -p 5 -q 2
```

To test the application, flows should be set up in the flow generator that match the values in the l3fwd_ipv4_route_array and/or l3fwd_ipv6_route_array table.

Please note that in order to test this application, the traffic generator should be generating valid fragmented IP packets. For IPv6, the only supported case is when no other extension headers other than fragment extension header are present in the packet.

The default l3fwd_ipv4_route_array table is:

```
struct l3fwd_ipv4_route l3fwd_ipv4_route_array[] = {
    {RTE_IPV4(100, 10, 0, 0), 16, 0},
    {RTE_IPV4(100, 20, 0, 0), 16, 1},
    {RTE_IPV4(100, 30, 0, 0), 16, 2},
    {RTE_IPV4(100, 40, 0, 0), 16, 3},
    {RTE_IPV4(100, 50, 0, 0), 16, 4},
    {RTE_IPV4(100, 60, 0, 0), 16, 5},
    {RTE_IPV4(100, 70, 0, 0), 16, 6},
    {RTE_IPV4(100, 80, 0, 0), 16, 7},
};
```

The default l3fwd_ipv6_route_array table is:

```
struct l3fwd_ipv6_route l3fwd_ipv6_route_array[] = {
    {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 0},
    {{2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 1},
    {{3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 2},
    {{4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 3},
    {{5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 4},
    {{6, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 5},
    {{7, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 6},
    {{8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 7},
};
```

For example, for the fragmented input IPv4 packet with destination address: 100.10.1.1, a reassembled IPv4 packet be sent out from port #0 to the destination address 100.10.1.1 once all the fragments are collected.

4.12.4 Explanation

The following sections provide some explanation of the sample application code. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. The following sections describe aspects that are specific to the IP reassemble sample application.

IPv4 Fragment Table Initialization

This application uses the rte_ip_frag library. Please refer to Programmer's Guide for more detailed explanation of how to use this library. Fragment table maintains information about already received fragments of the packet. Each IP packet is uniquely identified by triple <Source IP address>, <Destination IP address>, <ID>. To avoid lock contention, each RX queue has its own Fragment Table, e.g. the application can't handle the situation when different fragments of the same packet arrive through different RX queues. Each table entry can hold information about packet consisting of up to RTE_LIBRTE_IP_FRAG_MAX_FRAGS fragments.

```
frag_cycles = (rte_get_tsc_hz() + MS_PER_S - 1) / MS_PER_S * max_flow_ttl;

if ((qconf->frag_tbl[queue] = rte_ip_frag_tbl_create(max_flow_num, IPV4_FRAG_TBL_BUCKET_
↪ENTRIES, max_flow_num, frag_cycles, socket)) == NULL)
{
    RTE_LOG(ERR, IP_RSMBL, "ip_frag_tbl_create(%u) on " "lcore: %u for queue: %u failed\n", ␣
↪max_flow_num, lcore, queue);
    return -1;
}
```

Mempools Initialization

The reassembly application demands a lot of mbuf's to be allocated. At any given time up to (2 * max_flow_num * RTE_LIBRTE_IP_FRAG_MAX_FRAGS * <maximum number of mbufs per packet>) can be stored inside Fragment Table waiting for remaining fragments. To keep mempool size under reasonable limits and to avoid situation when one RX queue can starve other queues, each RX queue uses its own mempool.

```
nb_mbuf = RTE_MAX(max_flow_num, 2UL * MAX_PKT_BURST) * RTE_LIBRTE_IP_FRAG_MAX_FRAGS;
nb_mbuf *= (port_conf.rxmode.max_rx_pkt_len + BUF_SIZE - 1) / BUF_SIZE;
nb_mbuf *= 2; /* ipv4 and ipv6 */
nb_mbuf += RTE_TEST_RX_DESC_DEFAULT + RTE_TEST_TX_DESC_DEFAULT;
nb_mbuf = RTE_MAX(nb_mbuf, (uint32_t)NB_MBUF);

snprintf(buf, sizeof(buf), "mbuf_pool_%u_%u", lcore, queue);

if ((rxq->pool = rte_mempool_create(buf, nb_mbuf, MBUF_SIZE, 0, sizeof(struct rte_pktmbuf_pool_
↪private), rte_pktmbuf_pool_init, NULL,
    rte_pktmbuf_init, NULL, socket, MEMPOOL_F_SP_PUT | MEMPOOL_F_SC_GET)) == NULL) {

        RTE_LOG(ERR, IP_RSMBL, "mempool_create(%s) failed", buf);
        return -1;
}
```

Packet Reassembly and Forwarding

For each input packet, the packet forwarding operation is done by the l3fwd_simple_forward() function. If the packet is an IPv4 or IPv6 fragment, then it calls rte_ipv4_reassemble_packet() for IPv4 packets, or rte_ipv6_reassemble_packet() for IPv6 packets. These functions either return a pointer to valid mbuf that contains reassembled packet, or NULL (if the packet can't be reassembled for some reason). Then l3fwd_simple_forward() continues with the code for the packet forwarding decision (that is, the identification of the output interface for the packet) and actual transmit of the packet.

The rte_ipv4_reassemble_packet() or rte_ipv6_reassemble_packet() are responsible for:

1. Searching the Fragment Table for entry with packet's <IP Source Address, IP Destination Address, Packet ID>
2. If the entry is found, then check if that entry already timed-out. If yes, then free all previously received fragments, and remove information about them from the entry.
3. If no entry with such key is found, then try to create a new one by one of two ways:
    1. Use as empty entry
    2. Delete a timed-out entry, free mbufs associated with it mbufs and store a new entry with specified key in it.
4. Update the entry with new fragment information and check if a packet can be reassembled (the packet's entry contains all fragments).
    1. If yes, then, reassemble the packet, mark table's entry as empty and return the reassembled mbuf to the caller.
    2. If no, then just return a NULL to the caller.

If at any stage of packet processing a reassembly function encounters an error (can't insert new entry into the Fragment table, or invalid/timed-out fragment), then it will free all associated with the packet

fragments, mark the table entry as invalid and return NULL to the caller.

Debug logging and Statistics Collection

The RTE_LIBRTE_IP_FRAG_TBL_STAT controls statistics collection for the IP Fragment Table. This macro is disabled by default. To make ip_reassembly print the statistics to the standard output, the user must send either an USR1, INT or TERM signal to the process. For all of these signals, the ip_reassembly process prints Fragment table statistics for each RX queue, plus the INT and TERM will cause process termination as usual.

4.13 Kernel NIC Interface Sample Application

The Kernel NIC Interface (KNI) is a DPDK control plane solution that allows userspace applications to exchange packets with the kernel networking stack. To accomplish this, DPDK userspace applications use an IOCTL call to request the creation of a KNI virtual device in the Linux* kernel. The IOCTL call provides interface information and the DPDK's physical address space, which is re-mapped into the kernel address space by the KNI kernel loadable module that saves the information to a virtual device context. The DPDK creates FIFO queues for packet ingress and egress to the kernel module for each device allocated.

The KNI kernel loadable module is a standard net driver, which upon receiving the IOCTL call access the DPDK's FIFO queue to receive/transmit packets from/to the DPDK userspace application. The FIFO queues contain pointers to data packets in the DPDK. This:

- Provides a faster mechanism to interface with the kernel net stack and eliminates system calls
- Facilitates the DPDK using standard Linux* userspace net tools (tshark, rsync, and so on)
- Eliminate the copy_to_user and copy_from_user operations on packets.

The Kernel NIC Interface sample application is a simple example that demonstrates the use of the DPDK to create a path for packets to go through the Linux* kernel. This is done by creating one or more kernel net devices for each of the DPDK ports. The application allows the use of standard Linux tools (ethtool, iproute, tshark) with the DPDK ports and also the exchange of packets between the DPDK application and the Linux* kernel.

The Kernel NIC Interface sample application requires that the KNI kernel module `rte_kni` be loaded into the kernel. See *Kernel NIC Interface* for more information on loading the `rte_kni` kernel module.

4.13.1 Overview

The Kernel NIC Interface sample application `kni` allocates one or more KNI interfaces for each physical NIC port. For each physical NIC port, `kni` uses two DPDK threads in user space; one thread reads from the port and writes to the corresponding KNI interfaces and the other thread reads from the KNI interfaces and writes the data unmodified to the physical NIC port.

It is recommended to configure one KNI interface for each physical NIC port. The application can be configured with more than one KNI interface for each physical NIC port for performance testing or it can work together with VMDq support in future.

The packet flow through the Kernel NIC Interface application is as shown in the following figure.

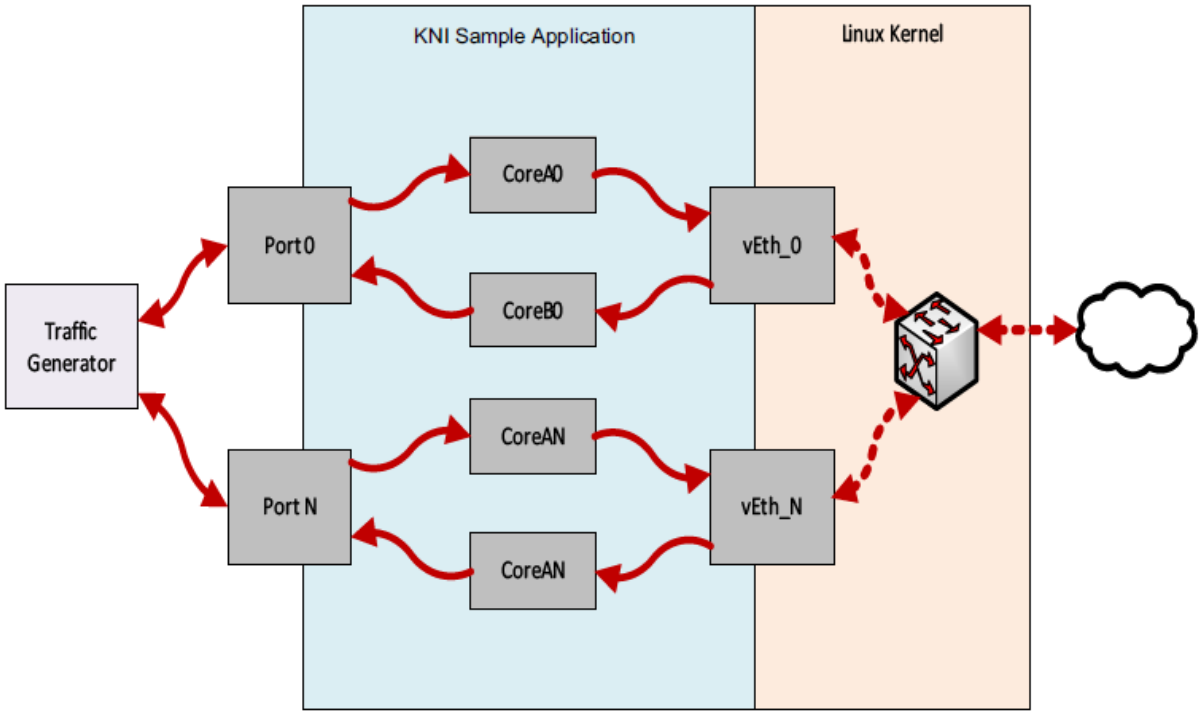


Fig. 4.1: Kernel NIC Application Packet Flow

If link monitoring is enabled with the `-m` command line flag, one additional pthread is launched which will check the link status of each physical NIC port and will update the carrier status of the corresponding KNI interface(s) to match the physical NIC port's state. This means that the KNI interface(s) will be disabled automatically when the Ethernet link goes down and enabled when the Ethernet link goes up.

If link monitoring is enabled, the `rte_kni` kernel module should be loaded such that the *default carrier state* is set to *off*. This ensures that the KNI interface is only enabled *after* the Ethernet link of the corresponding NIC port has reached the linkup state.

If link monitoring is not enabled, the `rte_kni` kernel module should be loaded with the *default carrier state* set to *on*. This sets the carrier state of the KNI interfaces to *on* when the KNI interfaces are enabled without regard to the actual link state of the corresponding NIC port. This is useful for testing in loopback mode where the NIC port may not be physically connected to anything.

4.13.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `examples/kni` sub-directory.

Note: This application is intended as a linux only.

4.13.3 Running the kni Example Application

The `kni` example application requires a number of command line options:

```
kni [EAL options] -- -p PORTMASK --config="(port,lcore_rx,lcore_tx[,lcore_kthread,...])[,(port,
↪lcore_rx,lcore_tx[,lcore_kthread,...])]" [-P] [-m]
```

Where:

- `-p PORTMASK`:

  Hexadecimal bitmask of ports to configure.

- `--config="(port,lcore_rx,lcore_tx[,lcore_kthread,...])[,(port,lcore_rx,lcore_tx[,lcore_kthread,...])]"`:

  Determines which lcores the Rx and Tx DPDK tasks, and (optionally) the KNI kernel thread(s) are bound to for each physical port.

- `-P`:

  Optional flag to set all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.

- `-m`:

  Optional flag to enable monitoring and updating of the Ethernet carrier state. With this option set, a thread will be started which will periodically check the Ethernet link status of the physical Ethernet ports and set the carrier state of the corresponding KNI network interface to match it. This means that the KNI interface will be disabled automatically when the Ethernet link goes down and enabled when the Ethernet link goes up.

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The `-c coremask` or `-l corelist` parameter of the EAL options must include the lcores specified by `lcore_rx` and `lcore_tx` for each port, but does not need to include lcores specified by `lcore_kthread` as those cores are used to pin the kernel threads in the `rte_kni` kernel module.

The `--config` parameter must include a set of `(port,lcore_rx,lcore_tx,[lcore_kthread,...])` values for each physical port specified in the `-p PORTMASK` parameter.

The optional `lcore_kthread` lcore ID parameter in `--config` can be specified zero, one or more times for each physical port.

If no lcore ID is specified for `lcore_kthread`, one KNI interface will be created for the physical port `port` and the KNI kernel thread(s) will have no specific core affinity.

If one or more lcore IDs are specified for `lcore_kthread`, a KNI interface will be created for each lcore ID specified, bound to the physical port `port`. If the `rte_kni` kernel module is loaded in *multiple kernel thread* mode, a kernel thread will be created for each KNI interface and bound to the specified core. If the `rte_kni` kernel module is loaded in *single kernel thread* mode, only one kernel thread is started for all KNI interfaces. The kernel thread will be bound to the first `lcore_kthread` lcore ID specified.

Example Configurations

The following commands will first load the `rte_kni` kernel module in *multiple kernel thread* mode. The `kni` application is then started using two ports; Port 0 uses lcore 4 for the Rx task, lcore 6 for the Tx task, and will create a single KNI interface `vEth0_0` with the kernel thread bound to lcore 8. Port 1 uses lcore 5 for the Rx task, lcore 7 for the Tx task, and will create a single KNI interface `vEth1_0` with the kernel thread bound to lcore 9.

```
# rmmod rte_kni
# insmod kmod/rte_kni.ko kthread_mode=multiple
# ./build/kni -l 4-7 -n 4 -- -P -p 0x3 -m --config="(0,4,6,8),(1,5,7,9)"
```

The following example is identical, except an additional `lcore_kthread` core is specified per physical port. In this case, `kni` will create four KNI interfaces: `vEth0_0`/`vEth0_1` bound to physical port 0 and `vEth1_0`/`vEth1_1` bound to physical port 1.

The kernel thread for each interface will be bound as follows:

- `vEth0_0` - bound to lcore 8.
- `vEth0_1` - bound to lcore 10.
- `vEth1_0` - bound to lcore 9.
- `vEth1_1` - bound to lcore 11

```
# rmmod rte_kni
# insmod kmod/rte_kni.ko kthread_mode=multiple
# ./build/kni -l 4-7 -n 4 -- -P -p 0x3 -m --config="(0,4,6,8,10),(1,5,7,9,11)"
```

The following example can be used to test the interface between the `kni` test application and the `rte_kni` kernel module. In this example, the `rte_kni` kernel module is loaded in *single kernel thread mode*, *loopback mode* enabled, and the *default carrier state* is set to *on* so that the corresponding physical NIC port does not have to be connected in order to use the KNI interface. One KNI interface `vEth0_0` is created for port 0 and one KNI interface `vEth1_0` is created for port 1. Since `rte_kni` is loaded in "single kernel thread" mode, the one kernel thread is bound to lcore 8.

Since the physical NIC ports are not being used, link monitoring can be disabled by **not** specifying the `-m` flag to `kni`:

```
# rmmod rte_kni
# insmod kmod/rte_kni.ko lo_mode=lo_mode_fifo carrier=on
# ./build/kni -l 4-7 -n 4 -- -P -p 0x3 --config="(0,4,6,8),(1,5,7,9)"
```

4.13.4 KNI Operations

Once the `kni` application is started, the user can use the normal Linux commands to manage the KNI interfaces as if they were any other Linux network interface.

Enable KNI interface and assign an IP address:

```
# ip addr add dev vEth0_0 192.168.0.1
```

Show KNI interface configuration and statistics:

```
# ip -s -d addr show vEth0_0
```

The user can also check and reset the packet statistics inside the `kni` application by sending the app the USR1 and USR2 signals:

```
# Print statistics
# pkill -USR1 kni

# Zero statistics
# pkill -USR2 kni
```

Dump network traffic:

```
# tshark -n -i vEth0_0
```

The normal Linux commands can also be used to change the MAC address and MTU size used by the physical NIC which corresponds to the KNI interface. However, if more than one KNI interface is configured for a physical port, these commands will only work on the first KNI interface for that port.

Change the MAC address:

```
# ip link set dev vEth0_0 lladdr 0C:01:02:03:04:08
```

Change the MTU size:

```
# ip link set dev vEth0_0 mtu 1450
```

Limited ethtool support:

```
# ethtool -i vEth0_0
```

When the `kni` application is closed, all the KNI interfaces are deleted from the Linux kernel.

4.13.5 Explanation

The following sections provide some explanation of code.

Initialization

Setup of mbuf pool, driver and queues is similar to the setup done in the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*.. In addition, one or more kernel NIC interfaces are allocated for each of the configured ports according to the command line parameters.

The code for allocating the kernel NIC interfaces for a specific port is in the function `kni_alloc`.

The other step in the initialization process that is unique to this sample application is the association of each port with lcores for RX, TX and kernel threads.

- One lcore to read from the port and write to the associated one or more KNI devices
- Another lcore to read from one or more KNI devices and write to the port
- Other lcores for pinning the kernel threads on one by one

This is done by using the `kni_port_params_array[]` array, which is indexed by the port ID. The code is in the function `parse_config`.

Packet Forwarding

After the initialization steps are completed, the main_loop() function is run on each lcore. This function first checks the lcore_id against the user provided lcore_rx and lcore_tx to see if this lcore is reading from or writing to kernel NIC interfaces.

For the case that reads from a NIC port and writes to the kernel NIC interfaces (`kni_ingress`), the packet reception is the same as in L2 Forwarding sample application (see *Receive, Process and Transmit Packets*). The packet transmission is done by sending mbufs into the kernel NIC interfaces by `rte_kni_tx_burst()`. The KNI library automatically frees the mbufs after the kernel successfully copied the mbufs.

For the other case that reads from kernel NIC interfaces and writes to a physical NIC port (`kni_egress`), packets are retrieved by reading mbufs from kernel NIC interfaces by `rte_kni_rx_burst()`. The packet transmission is the same as in the L2 Forwarding sample application (see *Receive, Process and Transmit Packets*).

4.14 Keep Alive Sample Application

The Keep Alive application is a simple example of a heartbeat/watchdog for packet processing cores. It demonstrates how to detect 'failed' DPDK cores and notify a fault management entity of this failure. Its purpose is to ensure the failure of the core does not result in a fault that is not detectable by a management entity.

4.14.1 Overview

The application demonstrates how to protect against 'silent outages' on packet processing cores. A Keep Alive Monitor Agent Core (master) monitors the state of packet processing cores (worker cores) by dispatching pings at a regular time interval (default is 5ms) and monitoring the state of the cores. Cores states are: Alive, MIA, Dead or Buried. MIA indicates a missed ping, and Dead indicates two missed pings within the specified time interval. When a core is Dead, a callback function is invoked to restart the packet processing core; A real life application might use this callback function to notify a higher level fault management entity of the core failure in order to take the appropriate corrective action.

Note: Only the worker cores are monitored. A local (on the host) mechanism or agent to supervise the Keep Alive Monitor Agent Core DPDK core is required to detect its failure.

Note: This application is based on the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. As such, the initialization and run-time paths are very similar to those of the L2 forwarding application.

4.14.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd_keep_alive` sub-directory.

4.14.3 Running the Application

The application has a number of command line options:

```
./build/l2fwd-keepalive [EAL options] \
        -- -p PORTMASK [-q NQ] [-K PERIOD] [-T PERIOD]
```

where,

- `p PORTMASK`: A hexadecimal bitmask of the ports to configure
- `q NQ`: A number of queues (=ports) per lcore (default is 1)
- `K PERIOD`: Heartbeat check period in ms(5ms default; 86400 max)
- `T PERIOD`: statistics will be refreshed each PERIOD seconds (0 to disable, 10 default, 86400 maximum).

To run the application in linux environment with 4 lcores, 16 ports 8 RX queues per lcore and a ping interval of 10ms, issue the command:

```
./build/l2fwd-keepalive -l 0-3 -n 4 -- -q 8 -p ffff -K 10
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.14.4 Explanation

The following sections provide some explanation of the The Keep-Alive/'Liveliness' conceptual scheme. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)*.

The Keep-Alive/'Liveliness' conceptual scheme:

- A Keep- Alive Agent Runs every N Milliseconds.
- DPDK Cores respond to the keep-alive agent.
- If keep-alive agent detects time-outs, it notifies the fault management entity through a callback function.

The following sections provide some explanation of the code aspects that are specific to the Keep Alive sample application.

The keepalive functionality is initialized with a struct rte_keepalive and the callback function to invoke in the case of a timeout.

```
rte_global_keepalive_info = rte_keepalive_create(&dead_core, NULL);
if (rte_global_keepalive_info == NULL)
    rte_exit(EXIT_FAILURE, "keepalive_create() failed");
```

The function that issues the pings keepalive_dispatch_pings() is configured to run every check_period milliseconds.

```
if (rte_timer_reset(&hb_timer,
        (check_period * rte_get_timer_hz()) / 1000,
        PERIODICAL,
        rte_lcore_id(),
```

(continues on next page)

(continued from previous page)

```
        &rte_keepalive_dispatch_pings,
        rte_global_keepalive_info
        ) != 0 )
    rte_exit(EXIT_FAILURE, "Keepalive setup failure.\n");
```

The rest of the initialization and run-time path follows the same paths as the L2 forwarding application. The only addition to the main processing loop is the mark alive functionality and the example random failures.

```
rte_keepalive_mark_alive(&rte_global_keepalive_info);
cur_tsc = rte_rdtsc();

/* Die randomly within 7 secs for demo purposes.. */
if (cur_tsc - tsc_initial > tsc_lifetime)
break;
```

The rte_keepalive_mark_alive function simply sets the core state to alive.

```
static inline void
rte_keepalive_mark_alive(struct rte_keepalive *keepcfg)
{
    keepcfg->live_data[rte_lcore_id()].core_state = RTE_KA_STATE_ALIVE;
}
```

4.15 Packet copying using Intel® QuickData Technology

4.15.1 Overview

This sample is intended as a demonstration of the basic components of a DPDK forwarding application and example of how to use IOAT driver API to make packets copies.

Also while forwarding, the MAC addresses are affected as follows:

- The source MAC address is replaced by the TX port MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

This application can be used to compare performance of using software packet copy with copy done using a DMA device for different sizes of packets. The example will print out statistics each second. The stats shows received/send packets and packets dropped or failed to copy.

4.15.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ioat` sub-directory.

4.15.3 Running the Application

In order to run the hardware copy application, the copying device needs to be bound to user-space IO driver.

Refer to the "IOAT Rawdev Driver" chapter in the "Rawdev Drivers" document for information on using the driver.

The application requires a number of command line options:

```
./build/ioatfwd [EAL options] -- [-p MASK] [-q NQ] [-s RS] [-c <sw|hw>]
    [--[no-]mac-updating]
```

where,

- p MASK: A hexadecimal bitmask of the ports to configure (default is all)
- q NQ: Number of Rx queues used per port equivalent to CBDMA channels per port (default is 1)
- c CT: Performed packet copy type: software (sw) or hardware using DMA (hw) (default is hw)
- s RS: Size of IOAT rawdev ring for hardware copy mode or rte_ring for software copy mode (default is 2048)
- –[no-]mac-updating: Whether MAC address of packets should be changed or not (default is mac-updating)

The application can be launched in various configurations depending on provided parameters. The app can use up to 2 lcores: one of them receives incoming traffic and makes a copy of each packet. The second lcore then updates MAC address and sends the copy. If one lcore per port is used, both operations are done sequentially. For each configuration an additional lcore is needed since the master lcore does not handle traffic but is responsible for configuration, statistics printing and safe shutdown of all ports and devices.

The application can use a maximum of 8 ports.

To run the application in a Linux environment with 3 lcores (the master lcore, plus two forwarding cores), a single port (port 0), software copying and MAC updating issue the command:

```
$ ./build/ioatfwd -l 0-2 -n 2 -- -p 0x1 --mac-updating -c sw
```

To run the application in a Linux environment with 2 lcores (the master lcore, plus one forwarding core), 2 ports (ports 0 and 1), hardware copying and no MAC updating issue the command:

```
$ ./build/ioatfwd -l 0-1 -n 1 -- -p 0x3 --no-mac-updating -c hw
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.15.4 Explanation

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with `rte_` and are explained in detail in the *DPDK API Documentation*.

The Main Function

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
/* init EAL */
ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");
```

The `main()` also allocates a mempool to hold the mbufs (Message Buffers) used by the application:

```
nb_mbufs = RTE_MAX(rte_eth_dev_count_avail() * (nb_rxd + nb_txd
    + MAX_PKT_BURST + rte_lcore_count() * MEMPOOL_CACHE_SIZE),
    MIN_POOL_SIZE);

/* Create the mbuf pool */
ioat_pktmbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", nb_mbufs,
    MEMPOOL_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE,
    rte_socket_id());
if (ioat_pktmbuf_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot init mbuf pool\n");
```

Mbufs are the packet buffer structure used by DPDK. They are explained in detail in the "Mbuf Library" section of the *DPDK Programmer's Guide*.

The `main()` function also initializes the ports:

```
/* Initialise each port */
RTE_ETH_FOREACH_DEV(portid) {
    port_init(portid, ioat_pktmbuf_pool);
}
```

Each port is configured using `port_init()` function. The Ethernet ports are configured with local settings using the `rte_eth_dev_configure()` function and the `port_conf` struct. The RSS is enabled so that multiple Rx queues could be used for packet receiving and copying by multiple CBDMA channels per port:

```
/* configuring port to use RSS for multiple RX queues */
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .mq_mode        = ETH_MQ_RX_RSS,
        .max_rx_pkt_len = RTE_ETHER_MAX_LEN
    },
    .rx_adv_conf = {
        .rss_conf = {
            .rss_key = NULL,
            .rss_hf = ETH_RSS_PROTO_MASK,
        }
```

(continues on next page)

(continued from previous page)

```
    }
};
```

For this example the ports are set up with the number of Rx queues provided with -q option and 1 Tx queue using the `rte_eth_rx_queue_setup()` and `rte_eth_tx_queue_setup()` functions.

The Ethernet port is then started:

```
ret = rte_eth_dev_start(portid);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_dev_start:err=%d, port=%u\n",
        ret, portid);
```

Finally the Rx port is set in promiscuous mode:

```
rte_eth_promiscuous_enable(portid);
```

After that each port application assigns resources needed.

```
check_link_status(ioat_enabled_port_mask);

if (!cfg.nb_ports) {
    rte_exit(EXIT_FAILURE,
        "All available ports are disabled. Please set portmask.\n");
}

/* Check if there is enough lcores for all ports. */
cfg.nb_lcores = rte_lcore_count() - 1;
if (cfg.nb_lcores < 1)
    rte_exit(EXIT_FAILURE,
        "There should be at least one slave lcore.\n");

ret = 0;

if (copy_mode == COPY_MODE_IOAT_NUM) {
    assign_rawdevs();
} else /* copy_mode == COPY_MODE_SW_NUM */ {
    assign_rings();
}
```

Depending on mode set (whether copy should be done by software or by hardware) special structures are assigned to each port. If software copy was chosen, application have to assign ring structures for packet exchanging between lcores assigned to ports.

```
static void
assign_rings(void)
{
    uint32_t i;

    for (i = 0; i < cfg.nb_ports; i++) {
        char ring_name[20];

        snprintf(ring_name, 20, "rx_to_tx_ring_%u", i);
        /* Create ring for inter core communication */
        cfg.ports[i].rx_to_tx_ring = rte_ring_create(
                ring_name, ring_size,
                rte_socket_id(), RING_F_SP_ENQ);

        if (cfg.ports[i].rx_to_tx_ring == NULL)
```

(continues on next page)

(continued from previous page)

```
            rte_exit(EXIT_FAILURE, "%s\n",
                    rte_strerror(rte_errno));
    }
}
```

When using hardware copy each Rx queue of the port is assigned an IOAT device (`assign_rawdevs()`) using IOAT Rawdev Driver API functions:

```
static void
assign_rawdevs(void)
{
    uint16_t nb_rawdev = 0, rdev_id = 0;
    uint32_t i, j;

    for (i = 0; i < cfg.nb_ports; i++) {
        for (j = 0; j < cfg.ports[i].nb_queues; j++) {
            struct rte_rawdev_info rdev_info = { 0 };

            do {
                if (rdev_id == rte_rawdev_count())
                    goto end;
                rte_rawdev_info_get(rdev_id++, &rdev_info);
            } while (strcmp(rdev_info.driver_name,
                IOAT_PMD_RAWDEV_NAME_STR) != 0);

            cfg.ports[i].ioat_ids[j] = rdev_id - 1;
            configure_rawdev_queue(cfg.ports[i].ioat_ids[j]);
            ++nb_rawdev;
        }
    }
end:
    if (nb_rawdev < cfg.nb_ports * cfg.ports[0].nb_queues)
        rte_exit(EXIT_FAILURE,
            "Not enough IOAT rawdevs (%u) for all queues (%u).\n",
            nb_rawdev, cfg.nb_ports * cfg.ports[0].nb_queues);
    RTE_LOG(INFO, IOAT, "Number of used rawdevs: %u.\n", nb_rawdev);
}
```

The initialization of hardware device is done by `rte_rawdev_configure()` function using `rte_rawdev_info` struct. After configuration the device is started using `rte_rawdev_start()` function. Each of the above operations is done in `configure_rawdev_queue()`.

```
static void
configure_rawdev_queue(uint32_t dev_id)
{
    struct rte_ioat_rawdev_config dev_config = { .ring_size = ring_size };
    struct rte_rawdev_info info = { .dev_private = &dev_config };

    if (rte_rawdev_configure(dev_id, &info) != 0) {
        rte_exit(EXIT_FAILURE,
            "Error with rte_rawdev_configure()\n");
    }
    if (rte_rawdev_start(dev_id) != 0) {
        rte_exit(EXIT_FAILURE,
            "Error with rte_rawdev_start()\n");
    }
}
```

If initialization is successful, memory for hardware device statistics is allocated.

Finally `main()` function starts all packet handling lcores and starts printing stats in a loop on the master lcore. The application can be interrupted and closed using `Ctrl-C`. The master lcore waits for all slave processes to finish, deallocates resources and exits.

The processing lcores launching function are described below.

The Lcores Launching Functions

As described above, `main()` function invokes `start_forwarding_cores()` function in order to start processing for each lcore:

```
static void start_forwarding_cores(void)
{
    uint32_t lcore_id = rte_lcore_id();

    RTE_LOG(INFO, IOAT, "Entering %s on lcore %u\n",
            __func__, rte_lcore_id());

    if (cfg.nb_lcores == 1) {
        lcore_id = rte_get_next_lcore(lcore_id, true, true);
        rte_eal_remote_launch((lcore_function_t *)rxtx_main_loop,
            NULL, lcore_id);
    } else if (cfg.nb_lcores > 1) {
        lcore_id = rte_get_next_lcore(lcore_id, true, true);
        rte_eal_remote_launch((lcore_function_t *)rx_main_loop,
            NULL, lcore_id);

        lcore_id = rte_get_next_lcore(lcore_id, true, true);
        rte_eal_remote_launch((lcore_function_t *)tx_main_loop, NULL,
            lcore_id);
    }
}
```

The function launches Rx/Tx processing functions on configured lcores using `rte_eal_remote_launch()`. The configured ports, their number and number of assigned lcores are stored in user-defined `rxtx_transmission_config` struct:

```
struct rxtx_transmission_config {
    struct rxtx_port_config ports[RTE_MAX_ETHPORTS];
    uint16_t nb_ports;
    uint16_t nb_lcores;
};
```

The structure is initialized in 'main()' function with the values corresponding to ports and lcores configuration provided by the user.

The Lcores Processing Functions

For receiving packets on each port, the `ioat_rx_port()` function is used. The function receives packets on each configured Rx queue. Depending on the mode the user chose, it will enqueue packets to IOAT rawdev channels and then invoke copy process (hardware copy), or perform software copy of each packet using `pktmbuf_sw_copy()` function and enqueue them to an rte_ring:

```
/* Receive packets on one port and enqueue to IOAT rawdev or rte_ring. */
static void
ioat_rx_port(struct rxtx_port_config *rx_config)
```

(continues on next page)

(continued from previous page)

```
{
    uint32_t nb_rx, nb_enq, i, j;
    struct rte_mbuf *pkts_burst[MAX_PKT_BURST];
    for (i = 0; i < rx_config->nb_queues; i++) {

        nb_rx = rte_eth_rx_burst(rx_config->rxtx_port, i,
            pkts_burst, MAX_PKT_BURST);

        if (nb_rx == 0)
            continue;

        port_statistics.rx[rx_config->rxtx_port] += nb_rx;

        if (copy_mode == COPY_MODE_IOAT_NUM) {
            /* Perform packet hardware copy */
            nb_enq = ioat_enqueue_packets(pkts_burst,
                nb_rx, rx_config->ioat_ids[i]);
            if (nb_enq > 0)
                rte_ioat_do_copies(rx_config->ioat_ids[i]);
        } else {
            /* Perform packet software copy, free source packets */
            int ret;
            struct rte_mbuf *pkts_burst_copy[MAX_PKT_BURST];

            ret = rte_mempool_get_bulk(ioat_pktmbuf_pool,
                (void *)pkts_burst_copy, nb_rx);

            if (unlikely(ret < 0))
                rte_exit(EXIT_FAILURE,
                    "Unable to allocate memory.\n");

            for (j = 0; j < nb_rx; j++)
                pktmbuf_sw_copy(pkts_burst[j],
                    pkts_burst_copy[j]);

            rte_mempool_put_bulk(ioat_pktmbuf_pool,
                (void *)pkts_burst, nb_rx);

            nb_enq = rte_ring_enqueue_burst(
                rx_config->rx_to_tx_ring,
                (void *)pkts_burst_copy, nb_rx, NULL);

            /* Free any not enqueued packets. */
            rte_mempool_put_bulk(ioat_pktmbuf_pool,
                (void *)&pkts_burst_copy[nb_enq],
                nb_rx - nb_enq);
        }

        port_statistics.copy_dropped[rx_config->rxtx_port] +=
            (nb_rx - nb_enq);
    }
}
```

The packets are received in burst mode using `rte_eth_rx_burst()` function. When using hardware copy mode the packets are enqueued in copying device's buffer using `ioat_enqueue_packets()` which calls `rte_ioat_enqueue_copy()`. When all received packets are in the buffer the copy operations are started by calling `rte_ioat_do_copies()`. Function `rte_ioat_enqueue_copy()` operates on physical address of the packet. Structure `rte_mbuf` contains only physical address to start of the data buffer (`buf_iova`). Thus the address is adjusted by `addr_offset` value in order to get the address of `rearm_data` member of `rte_mbuf`. That way both the packet data and metadata can be copied in a

single operation. This method can be used because the mbufs are direct mbufs allocated by the apps. If another app uses external buffers, or indirect mbufs, then multiple copy operations must be used.

```
static uint32_t
ioat_enqueue_packets(struct rte_mbuf **pkts,
    uint32_t nb_rx, uint16_t dev_id)
{
    int ret;
    uint32_t i;
    struct rte_mbuf *pkts_copy[MAX_PKT_BURST];

    const uint64_t addr_offset = RTE_PTR_DIFF(pkts[0]->buf_addr,
        &pkts[0]->rearm_data);

    ret = rte_mempool_get_bulk(ioat_pktmbuf_pool,
            (void *)pkts_copy, nb_rx);

    if (unlikely(ret < 0))
        rte_exit(EXIT_FAILURE, "Unable to allocate memory.\n");

    for (i = 0; i < nb_rx; i++) {
        /* Perform data copy */
        ret = rte_ioat_enqueue_copy(dev_id,
            pkts[i]->buf_iova
                - addr_offset,
            pkts_copy[i]->buf_iova
                - addr_offset,
            rte_pktmbuf_data_len(pkts[i])
                + addr_offset,
            (uintptr_t)pkts[i],
            (uintptr_t)pkts_copy[i],
            0 /* nofence */);

        if (ret != 1)
            break;
    }

    ret = i;
    /* Free any not enqueued packets. */
    rte_mempool_put_bulk(ioat_pktmbuf_pool, (void *)&pkts[i], nb_rx - i);
    rte_mempool_put_bulk(ioat_pktmbuf_pool, (void *)&pkts_copy[i],
        nb_rx - i);

    return ret;
}
```

All completed copies are processed by `ioat_tx_port()` function. When using hardware copy mode the function invokes `rte_ioat_completed_copies()` on each assigned IOAT channel to gather copied packets. If software copy mode is used the function dequeues copied packets from the rte_ring. Then each packet MAC address is changed if it was enabled. After that copies are sent in burst mode using `` rte_eth_tx_burst()``.

```
/* Transmit packets from IOAT rawdev/rte_ring for one port. */
static void
ioat_tx_port(struct rxtx_port_config *tx_config)
{
    uint32_t i, j, nb_dq = 0;
    struct rte_mbuf *mbufs_src[MAX_PKT_BURST];
    struct rte_mbuf *mbufs_dst[MAX_PKT_BURST];

    for (i = 0; i < tx_config->nb_queues; i++) {
```

(continues on next page)

(continued from previous page)

```
        if (copy_mode == COPY_MODE_IOAT_NUM) {
            /* Deque the mbufs from IOAT device. */
            nb_dq = rte_ioat_completed_copies(
                tx_config->ioat_ids[i], MAX_PKT_BURST,
                (void *)mbufs_src, (void *)mbufs_dst);
        } else {
            /* Deque the mbufs from rx_to_tx_ring. */
            nb_dq = rte_ring_dequeue_burst(
                tx_config->rx_to_tx_ring, (void *)mbufs_dst,
                MAX_PKT_BURST, NULL);
        }

        if (nb_dq == 0)
            return;

        if (copy_mode == COPY_MODE_IOAT_NUM)
            rte_mempool_put_bulk(ioat_pktmbuf_pool,
                (void *)mbufs_src, nb_dq);

        /* Update macs if enabled */
        if (mac_updating) {
            for (j = 0; j < nb_dq; j++)
                update_mac_addrs(mbufs_dst[j],
                    tx_config->rxtx_port);
        }

        const uint16_t nb_tx = rte_eth_tx_burst(
            tx_config->rxtx_port, 0,
            (void *)mbufs_dst, nb_dq);

        port_statistics.tx[tx_config->rxtx_port] += nb_tx;

        /* Free any unsent packets. */
        if (unlikely(nb_tx < nb_dq))
            rte_mempool_put_bulk(ioat_pktmbuf_pool,
            (void *)&mbufs_dst[nb_tx],
                nb_dq - nb_tx);
    }
}
```

The Packet Copying Functions

In order to perform packet copy there is a user-defined function `pktmbuf_sw_copy()` used. It copies a whole packet by copying metadata from source packet to new mbuf, and then copying a data chunk of source packet. Both memory copies are done using `rte_memcpy()`:

```
static inline void
pktmbuf_sw_copy(struct rte_mbuf *src, struct rte_mbuf *dst)
{
    /* Copy packet metadata */
    rte_memcpy(&dst->rearm_data,
        &src->rearm_data,
        offsetof(struct rte_mbuf, cacheline1)
            - offsetof(struct rte_mbuf, rearm_data));

    /* Copy packet data */
    rte_memcpy(rte_pktmbuf_mtod(dst, char *),
        rte_pktmbuf_mtod(src, char *), src->data_len);
}
```

The metadata in this example is copied from `rearm_data` member of `rte_mbuf` struct up to `cacheline1`.

In order to understand why software packet copying is done as shown above please refer to the "Mbuf Library" section of the *DPDK Programmer's Guide*.

4.16 L2 Forwarding with Crypto Sample Application

The L2 Forwarding with Crypto (l2fwd-crypto) sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK), in conjunction with the Cryptodev library.

4.16.1 Overview

The L2 Forwarding with Crypto sample application performs a crypto operation (cipher/hash) specified by the user from command line (or using the default values), with a crypto device capable of doing that operation, for each packet that is received on a RX_PORT and performs L2 forwarding. The destination port is the adjacent port from the enabled portmask, that is, if the first four ports are enabled (portmask 0xf), ports 0 and 1 forward into each other, and ports 2 and 3 forward into each other. Also, if MAC addresses updating is enabled, the MAC addresses are affected as follows:

- The source MAC address is replaced by the TX_PORT MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

4.16.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd-crypt` sub-directory.

4.16.3 Running the Application

The application requires a number of command line options:

```
./build/l2fwd-crypto [EAL options] -- [-p PORTMASK] [-q NQ] [-s] [-T PERIOD] /
[--cdev_type HW/SW/ANY] [--chain HASH_CIPHER/CIPHER_HASH/CIPHER_ONLY/HASH_ONLY/AEAD] /
[--cipher_algo ALGO] [--cipher_op ENCRYPT/DECRYPT] [--cipher_key KEY] /
[--cipher_key_random_size SIZE] [--cipher_iv IV] [--cipher_iv_random_size SIZE] /
[--auth_algo ALGO] [--auth_op GENERATE/VERIFY] [--auth_key KEY] /
[--auth_key_random_size SIZE] [--auth_iv IV] [--auth_iv_random_size SIZE] /
[--aead_algo ALGO] [--aead_op ENCRYPT/DECRYPT] [--aead_key KEY] /
[--aead_key_random_size SIZE] [--aead_iv] [--aead_iv_random_size SIZE] /
[--aad AAD] [--aad_random_size SIZE] /
[--digest size SIZE] [--sessionless] [--cryptodev_mask MASK] /
[--mac-updating] [--no-mac-updating]
```

where,

- p PORTMASK: A hexadecimal bitmask of the ports to configure (default is all the ports)
- q NQ: A number of queues (=ports) per lcore (default is 1)
- s: manage all ports from single core

- T PERIOD: statistics will be refreshed each PERIOD seconds
  (0 to disable, 10 default, 86400 maximum)
- cdev_type: select preferred crypto device type: HW, SW or anything (ANY)
  (default is ANY)
- chain: select the operation chaining to perform: Cipher->Hash (CIPHER_HASH),
  Hash->Cipher (HASH_CIPHER), Cipher (CIPHER_ONLY), Hash (HASH_ONLY)
  or AEAD (AEAD)
  (default is Cipher->Hash)
- cipher_algo: select the ciphering algorithm (default is aes-cbc)
- cipher_op: select the ciphering operation to perform: ENCRYPT or DECRYPT
  (default is ENCRYPT)
- cipher_key: set the ciphering key to be used. Bytes has to be separated with ":"
- cipher_key_random_size: set the size of the ciphering key,
  which will be generated randomly.
  Note that if –cipher_key is used, this will be ignored.
- cipher_iv: set the cipher IV to be used. Bytes has to be separated with ":"
- cipher_iv_random_size: set the size of the cipher IV, which will be generated randomly.
  Note that if –cipher_iv is used, this will be ignored.
- auth_algo: select the authentication algorithm (default is sha1-hmac)
- auth_op: select the authentication operation to perform: GENERATE or VERIFY
  (default is GENERATE)
- auth_key: set the authentication key to be used. Bytes has to be separated with ":"
- auth_key_random_size: set the size of the authentication key,
  which will be generated randomly.
  Note that if –auth_key is used, this will be ignored.
- auth_iv: set the auth IV to be used. Bytes has to be separated with ":"
- auth_iv_random_size: set the size of the auth IV, which will be generated randomly.
  Note that if –auth_iv is used, this will be ignored.
- aead_algo: select the AEAD algorithm (default is aes-gcm)
- aead_op: select the AEAD operation to perform: ENCRYPT or DECRYPT
  (default is ENCRYPT)
- aead_key: set the AEAD key to be used. Bytes has to be separated with ":"
- aead_key_random_size: set the size of the AEAD key,
  which will be generated randomly.

Note that if –aead_key is used, this will be ignored.

- aead_iv: set the AEAD IV to be used. Bytes has to be separated with ":"
- aead_iv_random_size: set the size of the AEAD IV, which will be generated randomly.

  Note that if –aead_iv is used, this will be ignored.
- aad: set the AAD to be used. Bytes has to be separated with ":"
- aad_random_size: set the size of the AAD, which will be generated randomly.

  Note that if –aad is used, this will be ignored.
- digest_size: set the size of the digest to be generated/verified.
- sessionless: no crypto session will be created.
- cryptodev_mask: A hexadecimal bitmask of the cryptodevs to be used by the application.

  (default is all cryptodevs).
- [no-]mac-updating: Enable or disable MAC addresses updating (enabled by default).

The application requires that crypto devices capable of performing the specified crypto operation are available on application initialization. This means that HW crypto device/s must be bound to a DPDK driver or a SW crypto device/s (virtual crypto PMD) must be created (using –vdev).

To run the application in linux environment with 2 lcores, 2 ports and 2 crypto devices, issue the command:

```
$ ./build/l2fwd-crypto -l 0-1 -n 4 --vdev "crypto_aesni_mb0" \
--vdev "crypto_aesni_mb1" -- -p 0x3 --chain CIPHER_HASH \
--cipher_op ENCRYPT --cipher_algo aes-cbc \
--cipher_key 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f \
--auth_op GENERATE --auth_algo aes-xcbc-mac \
--auth_key 10:11:12:13:14:15:16:17:18:19:1a:1b:1c:1d:1e:1f
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

Note:

- The `l2fwd-crypto` sample application requires IPv4 packets for crypto operation.
- If multiple Ethernet ports is passed, then equal number of crypto devices are to be passed.
- All crypto devices shall use the same session.

4.16.4 Explanation

The L2 forward with Crypto application demonstrates the performance of a crypto operation on a packet received on a RX PORT before forwarding it to a TX PORT.

The following figure illustrates a sample flow of a packet in the application, from reception until transmission.

Fig. 4.2: Encryption flow Through the L2 Forwarding with Crypto Application

The following sections provide some explanation of the application.

Crypto operation specification

All the packets received in all the ports get transformed by the crypto device/s (ciphering and/or authentication). The crypto operation to be performed on the packet is parsed from the command line (go to "Running the Application" section for all the options).

If no parameter is passed, the default crypto operation is:

- Encryption with AES-CBC with 128 bit key.
- Authentication with SHA1-HMAC (generation).
- Keys, IV and AAD are generated randomly.

There are two methods to pass keys, IV and ADD from the command line:

- Passing the full key, separated bytes by ":":

```
--cipher_key 00:11:22:33:44
```

- Passing the size, so key is generated randomly:

```
--cipher_key_random_size 16
```

Note:
If full key is passed (first method) and the size is passed as well (second method), the latter will be ignored.

Size of these keys are checked (regardless the method), before starting the app, to make sure that it is supported by the crypto devices.

Crypto device initialization

Once the encryption operation is defined, crypto devices are initialized. The crypto devices must be either bound to a DPDK driver (if they are physical devices) or created using the EAL option –vdev (if they are virtual devices), when running the application.

The initialize_cryptodevs() function performs the device initialization. It iterates through the list of the available crypto devices and check which ones are capable of performing the operation. Each device has a set of capabilities associated with it, which are stored in the device info structure, so the function checks if the operation is within the structure of each device.

The following code checks if the device supports the specified cipher algorithm (similar for the authentication algorithm):

```
/* Check if device supports cipher algo */
i = 0;
opt_cipher_algo = options->cipher_xform.cipher.algo;
cap = &dev_info.capabilities[i];
while (cap->op != RTE_CRYPTO_OP_TYPE_UNDEFINED) {
        cap_cipher_algo = cap->sym.cipher.algo;
        if (cap->sym.xform_type ==
                        RTE_CRYPTO_SYM_XFORM_CIPHER) {
                if (cap_cipher_algo == opt_cipher_algo) {
                        if (check_type(options, &dev_info) == 0)
```

(continues on next page)

(continued from previous page)

```
                                break;
                }
        }
        cap = &dev_info.capabilities[++i];
}
```

If a capable crypto device is found, key sizes are checked to see if they are supported (cipher key and IV for the ciphering):

```
/*
 * Check if length of provided cipher key is supported
 * by the algorithm chosen.
 */
if (options->ckey_param) {
        if (check_supported_size(
                        options->cipher_xform.cipher.key.length,
                        cap->sym.cipher.key_size.min,
                        cap->sym.cipher.key_size.max,
                        cap->sym.cipher.key_size.increment)
                                != 0) {
                printf("Unsupported cipher key length\n");
                return -1;
        }
/*
 * Check if length of the cipher key to be randomly generated
 * is supported by the algorithm chosen.
 */
} else if (options->ckey_random_size != -1) {
        if (check_supported_size(options->ckey_random_size,
                        cap->sym.cipher.key_size.min,
                        cap->sym.cipher.key_size.max,
                        cap->sym.cipher.key_size.increment)
                                != 0) {
                printf("Unsupported cipher key length\n");
                return -1;
        }
        options->cipher_xform.cipher.key.length =
                                options->ckey_random_size;
/* No size provided, use minimum size. */
} else
        options->cipher_xform.cipher.key.length =
                        cap->sym.cipher.key_size.min;
```

After all the checks, the device is configured and it is added to the crypto device list.

Note:

The number of crypto devices that supports the specified crypto operation must be at least the number of ports to be used.

Session creation

The crypto operation has a crypto session associated to it, which contains information such as the transform chain to perform (e.g. ciphering then hashing), pointers to the keys, lengths... etc.

This session is created and is later attached to the crypto operation:

```
static struct rte_cryptodev_sym_session *
initialize_crypto_session(struct l2fwd_crypto_options *options,
                uint8_t cdev_id)
{
        struct rte_crypto_sym_xform *first_xform;
        struct rte_cryptodev_sym_session *session;
        uint8_t socket_id = rte_cryptodev_socket_id(cdev_id);
        struct rte_mempool *sess_mp = session_pool_socket[socket_id];


        if (options->xform_chain == L2FWD_CRYPTO_AEAD) {
                first_xform = &options->aead_xform;
        } else if (options->xform_chain == L2FWD_CRYPTO_CIPHER_HASH) {
                first_xform = &options->cipher_xform;
                first_xform->next = &options->auth_xform;
        } else if (options->xform_chain == L2FWD_CRYPTO_HASH_CIPHER) {
                first_xform = &options->auth_xform;
                first_xform->next = &options->cipher_xform;
        } else if (options->xform_chain == L2FWD_CRYPTO_CIPHER_ONLY) {
                first_xform = &options->cipher_xform;
        } else {
                first_xform = &options->auth_xform;
        }

        session = rte_cryptodev_sym_session_create(sess_mp);

        if (session == NULL)
                return NULL;

       if (rte_cryptodev_sym_session_init(cdev_id, session,
                            first_xform, sess_mp) < 0)
                return NULL;

       return session;
}

...

port_cparams[i].session = initialize_crypto_session(options,
                              port_cparams[i].dev_id);
```

Crypto operation creation

Given N packets received from a RX PORT, N crypto operations are allocated and filled:

```
if (nb_rx) {
/*
 * If we can't allocate a crypto_ops, then drop
 * the rest of the burst and dequeue and
 * process the packets to free offload structs
 */
if (rte_crypto_op_bulk_alloc(
```

(continues on next page)

(continued from previous page)

```
                l2fwd_crypto_op_pool,
                RTE_CRYPTO_OP_TYPE_SYMMETRIC,
                ops_burst, nb_rx) !=
                                nb_rx) {
        for (j = 0; j < nb_rx; j++)
                rte_pktmbuf_free(pkts_burst[i]);

        nb_rx = 0;
}
```

After filling the crypto operation (including session attachment), the mbuf which will be transformed is attached to it:

```
op->sym->m_src = m;
```

Since no destination mbuf is set, the source mbuf will be overwritten after the operation is done (in-place).

Crypto operation enqueuing/dequeuing

Once the operation has been created, it has to be enqueued in one of the crypto devices. Before doing so, for performance reasons, the operation stays in a buffer. When the buffer has enough operations (MAX_PKT_BURST), they are enqueued in the device, which will perform the operation at that moment:

```
static int
l2fwd_crypto_enqueue(struct rte_crypto_op *op,
                struct l2fwd_crypto_params *cparams)
{
        unsigned lcore_id, len;
        struct lcore_queue_conf *qconf;

        lcore_id = rte_lcore_id();

        qconf = &lcore_queue_conf[lcore_id];
        len = qconf->op_buf[cparams->dev_id].len;
        qconf->op_buf[cparams->dev_id].buffer[len] = op;
        len++;

        /* enough ops to be sent */
        if (len == MAX_PKT_BURST) {
                l2fwd_crypto_send_burst(qconf, MAX_PKT_BURST, cparams);
                len = 0;
        }

        qconf->op_buf[cparams->dev_id].len = len;
        return 0;
}

...

static int
l2fwd_crypto_send_burst(struct lcore_queue_conf *qconf, unsigned n,
                struct l2fwd_crypto_params *cparams)
{
        struct rte_crypto_op **op_buffer;
        unsigned ret;

        op_buffer = (struct rte_crypto_op **)
                        qconf->op_buf[cparams->dev_id].buffer;
```

(continues on next page)

(continued from previous page)

```
        ret = rte_cryptodev_enqueue_burst(cparams->dev_id,
                        cparams->qp_id, op_buffer, (uint16_t) n);

        crypto_statistics[cparams->dev_id].enqueued += ret;
        if (unlikely(ret < n)) {
                crypto_statistics[cparams->dev_id].errors += (n - ret);
                do {
                        rte_pktmbuf_free(op_buffer[ret]->sym->m_src);
                        rte_crypto_op_free(op_buffer[ret]);
                } while (++ret < n);
        }

        return 0;
}
```

After this, the operations are dequeued from the device, and the transformed mbuf is extracted from the operation. Then, the operation is freed and the mbuf is forwarded as it is done in the L2 forwarding application.

```
/* Dequeue packets from Crypto device */
do {
        nb_rx = rte_cryptodev_dequeue_burst(
                        cparams->dev_id, cparams->qp_id,
                        ops_burst, MAX_PKT_BURST);

        crypto_statistics[cparams->dev_id].dequeued +=
                        nb_rx;

        /* Forward crypto'd packets */
        for (j = 0; j < nb_rx; j++) {
                m = ops_burst[j]->sym->m_src;

                rte_crypto_op_free(ops_burst[j]);
                l2fwd_simple_forward(m, portid);
        }
} while (nb_rx == MAX_PKT_BURST);
```

4.17 L2 Forwarding Sample Application (in Real and Virtualized Environments) with core load statistics.

The L2 Forwarding sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) which also takes advantage of Single Root I/O Virtualization (SR-IOV) features in a virtualized environment.

Note: This application is a variation of L2 Forwarding sample application. It demonstrate possible scheme of job stats library usage therefore some parts of this document is identical with original L2 forwarding application.

4.17.1 Overview

The L2 Forwarding sample application, which can operate in real and virtualized environments, performs L2 forwarding for each packet that is received. The destination port is the adjacent port from the enabled portmask, that is, if the first four ports are enabled (portmask 0xf), ports 1 and 2 forward into each other, and ports 3 and 4 forward into each other. Also, the MAC addresses are affected as follows:

- The source MAC address is replaced by the TX port MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

This application can be used to benchmark performance using a traffic-generator, as shown in the Fig. 4.3.

The application can also be used in a virtualized environment as shown in Fig. 4.4.

The L2 Forwarding application can also be used as a starting point for developing a new application based on the DPDK.

Fig. 4.3: Performance Benchmark Setup (Basic Environment)

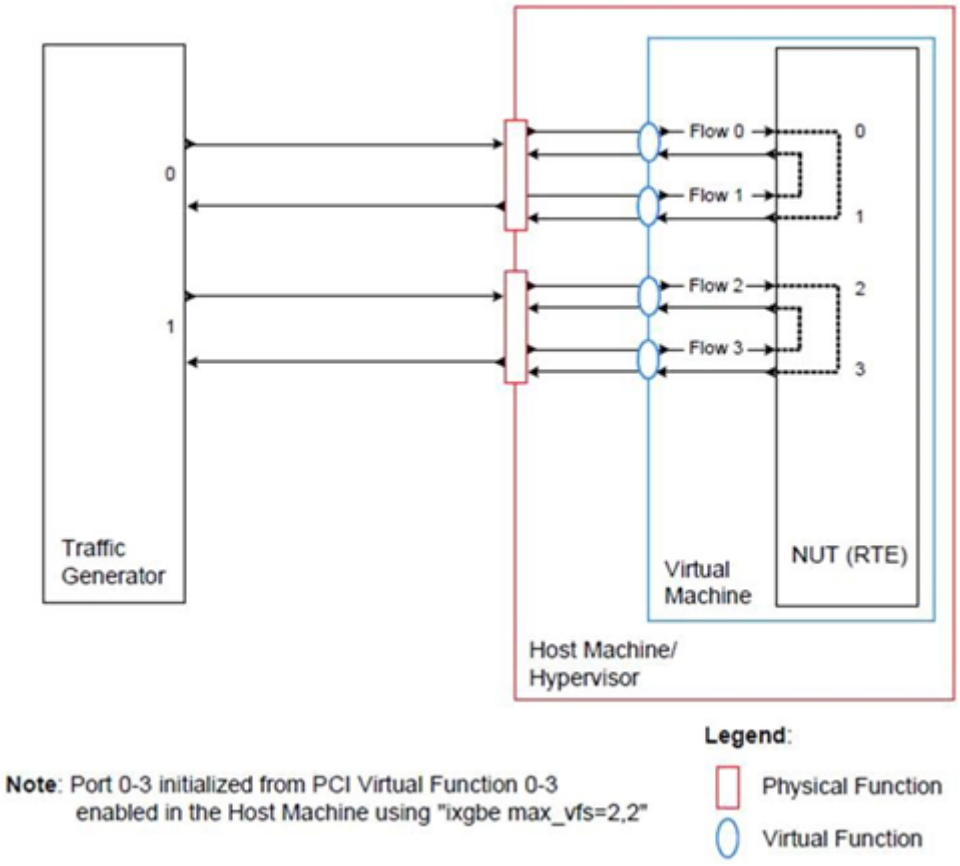


Fig. 4.4: Performance Benchmark Setup (Virtualized Environment)

Virtual Function Setup Instructions

This application can use the virtual function available in the system and therefore can be used in a virtual machine without passing through the whole Network Device into a guest machine in a virtualized scenario. The virtual functions can be enabled in the host machine or the hypervisor with the respective physical function driver.

For example, in a Linux* host machine, it is possible to enable a virtual function using the following command:

```
modprobe ixgbe max_vfs=2,2
```

This command enables two Virtual Functions on each of Physical Function of the NIC, with two physical ports in the PCI configuration space. It is important to note that enabled Virtual Function 0 and 2 would belong to Physical Function 0 and Virtual Function 1 and 3 would belong to Physical Function 1, in this case enabling a total of four Virtual Functions.

4.17.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd-jobstats` sub-directory.

4.17.3 Running the Application

The application requires a number of command line options:

```
./build/l2fwd-jobstats [EAL options] -- -p PORTMASK [-q NQ] [-l]
```

where,

- p PORTMASK: A hexadecimal bitmask of the ports to configure
- q NQ: A number of queues (=ports) per lcore (default is 1)
- l: Use locale thousands separator when formatting big numbers.

To run the application in linux environment with 4 lcores, 16 ports, 8 RX queues per lcore and thousands separator printing, issue the command:

```
$ ./build/l2fwd-jobstats -l 0-3 -n 4 -- -q 8 -p ffff -l
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.17.4 Explanation

The following sections provide some explanation of the code.

Command Line Arguments

The L2 Forwarding sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments (see *Running the Application*). The preferred way to parse parameters is to use the getopt() function, since it is part of a well-defined and portable library.

The parsing of arguments is done in the l2fwd_parse_args() function. The method of argument parsing is not described here. Refer to the *glibc getopt(3)* man page for details.

EAL arguments are parsed first, then application-specific arguments. This is done at the beginning of the main() function:

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");

argc -= ret;
argv += ret;

/* parse application arguments (after the EAL ones) */

ret = l2fwd_parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid L2FWD arguments\n");
```

Mbuf Pool Initialization

Once the arguments are parsed, the mbuf pool is created. The mbuf pool contains a set of mbuf objects that will be used by the driver and the application to store network packet data:

```
/* create the mbuf pool */
l2fwd_pktmbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", NB_MBUF,
            MEMPOOL_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE,
            rte_socket_id());

if (l2fwd_pktmbuf_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot init mbuf pool\n");
```

The rte_mempool is a generic structure used to handle pools of objects. In this case, it is necessary to create a pool that will be used by the driver. The number of allocated pkt mbufs is NB_MBUF, with a data room size of RTE_MBUF_DEFAULT_BUF_SIZE each. A per-lcore cache of MEMPOOL_CACHE_SIZE mbufs is kept. The memory is allocated in rte_socket_id() socket, but it is possible to extend this code to allocate one mbuf pool per socket.

The rte_pktmbuf_pool_create() function uses the default mbuf pool and mbuf initializers, respectively rte_pktmbuf_pool_init() and rte_pktmbuf_init(). An advanced application may want to use the mempool API to create the mbuf pool with more control.

Driver Initialization

The main part of the code in the main() function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the Poll Mode Driver in the *DPDK Programmer's Guide* and the *DPDK API Reference*.

```
/* reset l2fwd_dst_ports */

for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++)
    l2fwd_dst_ports[portid] = 0;

last_port = 0;

/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */
RTE_ETH_FOREACH_DEV(portid) {
    /* skip ports that are not enabled */
    if ((l2fwd_enabled_port_mask & (1 << portid)) == 0)
       continue;

    if (nb_ports_in_mask % 2) {
        l2fwd_dst_ports[portid] = last_port;
        l2fwd_dst_ports[last_port] = portid;
    }
    else
       last_port = portid;

    nb_ports_in_mask++;

    rte_eth_dev_info_get((uint8_t) portid, &dev_info);
}
```

The next step is to configure the RX and TX queues. For each port, there is only one RX queue (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. The rte_eth_dev_configure() function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t)portid, 1, 1, &port_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot configure device: "
        "err=%d, port=%u\n",
        ret, portid);
```

RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the -q option, which specifies the number of queues per lcore.

For example, if the user specifies -q 4, the application is able to poll four ports with one lcore. If there are 16 ports on the target (and if the portmask argument is -p ffff), the application will need four lcores to poll all the ports.

```
ret = rte_eth_rx_queue_setup(portid, 0, nb_rxd,
             rte_eth_dev_socket_id(portid),
             NULL,
             l2fwd_pktmbuf_pool);

if (ret < 0)
```

(continues on next page)

(continued from previous page)

```
    rte_exit(EXIT_FAILURE, "rte_eth_rx_queue_setup:err=%d, port=%u\n",
            ret, (unsigned) portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called struct lcore_queue_conf.

```
struct lcore_queue_conf {
    unsigned n_rx_port;
    unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE];
    truct mbuf_table tx_mbufs[RTE_MAX_ETHPORTS];

    struct rte_timer rx_timers[MAX_RX_QUEUE_PER_LCORE];
    struct rte_jobstats port_fwd_jobs[MAX_RX_QUEUE_PER_LCORE];

    struct rte_timer flush_timer;
    struct rte_jobstats flush_job;
    struct rte_jobstats idle_job;
    struct rte_jobstats_context jobs_context;

    rte_atomic16_t stats_read_pending;
    rte_spinlock_t lock;
} __rte_cache_aligned;
```

Values of struct lcore_queue_conf:

- n_rx_port and rx_port_list[] are used in the main packet processing loop (see Section *Receive, Process and Transmit Packets* later in this chapter).
- rx_timers and flush_timer are used to ensure forced TX on low packet rate.
- flush_job, idle_job and jobs_context are librte_jobstats objects used for managing l2fwd jobs.
- stats_read_pending and lock are used during job stats read phase.

TX Queue Initialization

Each lcore should be able to transmit on any port. For every port, a single TX queue is initialized.

```
/* init one TX queue on each port */

fflush(stdout);
ret = rte_eth_tx_queue_setup(portid, 0, nb_txd,
        rte_eth_dev_socket_id(portid),
        NULL);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup:err=%d, port=%u\n",
            ret, (unsigned) portid);
```

Jobs statistics initialization

There are several statistics objects available:

- Flush job statistics

```
rte_jobstats_init(&qconf->flush_job, "flush", drain_tsc, drain_tsc,
        drain_tsc, 0);

rte_timer_init(&qconf->flush_timer);
ret = rte_timer_reset(&qconf->flush_timer, drain_tsc, PERIODICAL,
            lcore_id, &l2fwd_flush_job, NULL);

if (ret < 0) {
    rte_exit(1, "Failed to reset flush job timer for lcore %u: %s",
                lcore_id, rte_strerror(-ret));
}
```

- Statistics per RX port

```
rte_jobstats_init(job, name, 0, drain_tsc, 0, MAX_PKT_BURST);
rte_jobstats_set_update_period_function(job, l2fwd_job_update_cb);

rte_timer_init(&qconf->rx_timers[i]);
ret = rte_timer_reset(&qconf->rx_timers[i], 0, PERIODICAL, lcore_id,
        l2fwd_fwd_job, (void *)(uintptr_t)i);

if (ret < 0) {
    rte_exit(1, "Failed to reset lcore %u port %u job timer: %s",
                lcore_id, qconf->rx_port_list[i], rte_strerror(-ret));
}
```

Following parameters are passed to rte_jobstats_init():

- 0 as minimal poll period
- drain_tsc as maximum poll period
- MAX_PKT_BURST as desired target value (RX burst size)

Main loop

The forwarding path is reworked comparing to original L2 Forwarding application. In the l2fwd_main_loop() function three loops are placed.

```
for (;;) {
    rte_spinlock_lock(&qconf->lock);

    do {
        rte_jobstats_context_start(&qconf->jobs_context);

        /* Do the Idle job:
         * - Read stats_read_pending flag
         * - check if some real job need to be executed
         */
        rte_jobstats_start(&qconf->jobs_context, &qconf->idle_job);

        do {
            uint8_t i;
            uint64_t now = rte_get_timer_cycles();
```

(continues on next page)

(continued from previous page)

```
            need_manage = qconf->flush_timer.expire < now;
            /* Check if we was esked to give a stats. */
            stats_read_pending =
                    rte_atomic16_read(&qconf->stats_read_pending);
            need_manage |= stats_read_pending;

            for (i = 0; i < qconf->n_rx_port && !need_manage; i++)
                need_manage = qconf->rx_timers[i].expire < now;

        } while (!need_manage);
        rte_jobstats_finish(&qconf->idle_job, qconf->idle_job.target);

        rte_timer_manage();
        rte_jobstats_context_finish(&qconf->jobs_context);
    } while (likely(stats_read_pending == 0));

    rte_spinlock_unlock(&qconf->lock);
    rte_pause();
}
```

First infinite for loop is to minimize impact of stats reading. Lock is only locked/unlocked when asked.

Second inner while loop do the whole jobs management. When any job is ready, the use rte_timer_manage() is used to call the job handler. In this place functions l2fwd_fwd_job() and l2fwd_flush_job() are called when needed. Then rte_jobstats_context_finish() is called to mark loop end - no other jobs are ready to execute. By this time stats are ready to be read and if stats_read_pending is set, loop breaks allowing stats to be read.

Third do-while loop is the idle job (idle stats counter). Its only purpose is monitoring if any job is ready or stats job read is pending for this lcore. Statistics from this part of code is considered as the headroom available for additional processing.

Receive, Process and Transmit Packets

The main task of l2fwd_fwd_job() function is to read ingress packets from the RX queue of particular port and forward it. This is done using the following code:

```
total_nb_rx = rte_eth_rx_burst((uint8_t) portid, 0, pkts_burst,
        MAX_PKT_BURST);

for (j = 0; j < total_nb_rx; j++) {
    m = pkts_burst[j];
    rte_prefetch0(rte_pktmbuf_mtod(m, void *));
    l2fwd_simple_forward(m, portid);
}
```

Packets are read in a burst of size MAX_PKT_BURST. Then, each mbuf in the table is processed by the l2fwd_simple_forward() function. The processing is very simple: process the TX port from the RX port, then replace the source and destination MAC addresses.

The rte_eth_rx_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

After first read second try is issued.

```
if (total_nb_rx == MAX_PKT_BURST) {
    const uint16_t nb_rx = rte_eth_rx_burst((uint8_t) portid, 0, pkts_burst,
            MAX_PKT_BURST);

    total_nb_rx += nb_rx;
    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
        rte_prefetch0(rte_pktmbuf_mtod(m, void *));
        l2fwd_simple_forward(m, portid);
    }
}
```

This second read is important to give job stats library a feedback how many packets was processed.

```
/* Adjust period time in which we are running here. */
if (rte_jobstats_finish(job, total_nb_rx) != 0) {
    rte_timer_reset(&qconf->rx_timers[port_idx], job->period, PERIODICAL,
            lcore_id, l2fwd_fwd_job, arg);
}
```

To maximize performance exactly MAX_PKT_BURST is expected (the target value) to be read for each l2fwd_fwd_job() call. If total_nb_rx is smaller than target value job->period will be increased. If it is greater the period will be decreased.

Note: In the following code, one line for getting the output port requires some explanation.

During the initialization process, a static array of destination ports (l2fwd_dst_ports[]) is filled such that for each source port, a destination port is assigned that is either the next or previous enabled port from the portmask. Naturally, the number of ports in the portmask must be even, otherwise, the application exits.

```
static void
l2fwd_simple_forward(struct rte_mbuf *m, unsigned portid)
{
    struct rte_ether_hdr *eth;
    void *tmp;
    unsigned dst_port;

    dst_port = l2fwd_dst_ports[portid];

    eth = rte_pktmbuf_mtod(m, struct rte_ether_hdr *);

    /* 02:00:00:00:00:xx */

    tmp = &eth->d_addr.addr_bytes[0];

    *((uint64_t *)tmp) = 0x000000000002 + ((uint64_t) dst_port << 40);

    /* src addr */

    rte_ether_addr_copy(&l2fwd_ports_eth_addr[dst_port], &eth->s_addr);

    l2fwd_send_packet(m, (uint8_t) dst_port);
}
```

Then, the packet is sent using the l2fwd_send_packet (m, dst_port) function. For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the l2fwd_send_burst() function directly from the main loop to send all the received

packets on the same TX port, using the burst-oriented send function, which is more efficient.

However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that, so the same approach can be reused in a more complex application.

The l2fwd_send_packet() function stores the packet in a per-lcore and per-txport table. If the table is full, the whole packets table is transmitted using the l2fwd_send_burst() function:

```
/* Send the packet on an output interface */

static int
l2fwd_send_packet(struct rte_mbuf *m, uint16_t port)
{
    unsigned lcore_id, len;
    struct lcore_queue_conf *qconf;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = m;
    len++;

    /* enough pkts to be sent */

    if (unlikely(len == MAX_PKT_BURST)) {
        l2fwd_send_burst(qconf, MAX_PKT_BURST, port);
        len = 0;
    }

    qconf->tx_mbufs[port].len = len; return 0;
}
```

To ensure that no packets remain in the tables, the flush job exists. The l2fwd_flush_job() is called periodically to for each lcore draining TX queue of each port. This technique introduces some latency when there are not many packets to send, however it improves performance:

```
static void
l2fwd_flush_job(__rte_unused struct rte_timer *timer, __rte_unused void *arg)
{
    uint64_t now;
    unsigned lcore_id;
    struct lcore_queue_conf *qconf;
    struct mbuf_table *m_table;
    uint16_t portid;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];

    rte_jobstats_start(&qconf->jobs_context, &qconf->flush_job);

    now = rte_get_timer_cycles();
    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
        m_table = &qconf->tx_mbufs[portid];
        if (m_table->len == 0 || m_table->next_flush_time <= now)
            continue;

        l2fwd_send_burst(qconf, portid);
    }
```

(continues on next page)

(continued from previous page)

```
    /* Pass target to indicate that this job is happy of time interval
     * in which it was called. */
    rte_jobstats_finish(&qconf->flush_job, qconf->flush_job.target);
}
```

4.18 L2 Forwarding Sample Application (in Real and Virtualized Environments)

The L2 Forwarding sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) which also takes advantage of Single Root I/O Virtualization (SR-IOV) features in a virtualized environment.

Note: Please note that previously a separate L2 Forwarding in Virtualized Environments sample application was used, however, in later DPDK versions these sample applications have been merged.

4.18.1 Overview

The L2 Forwarding sample application, which can operate in real and virtualized environments, performs L2 forwarding for each packet that is received on an RX_PORT. The destination port is the adjacent port from the enabled portmask, that is, if the first four ports are enabled (portmask 0xf), ports 1 and 2 forward into each other, and ports 3 and 4 forward into each other. Also, if MAC addresses updating is enabled, the MAC addresses are affected as follows:

- The source MAC address is replaced by the TX_PORT MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

This application can be used to benchmark performance using a traffic-generator, as shown in the Fig. 4.5, or in a virtualized environment as shown in Fig. 4.6.

Fig. 4.5: Performance Benchmark Setup (Basic Environment)

This application may be used for basic VM to VM communication as shown in Fig. 4.7, when MAC addresses updating is disabled.

The L2 Forwarding application can also be used as a starting point for developing a new application based on the DPDK.

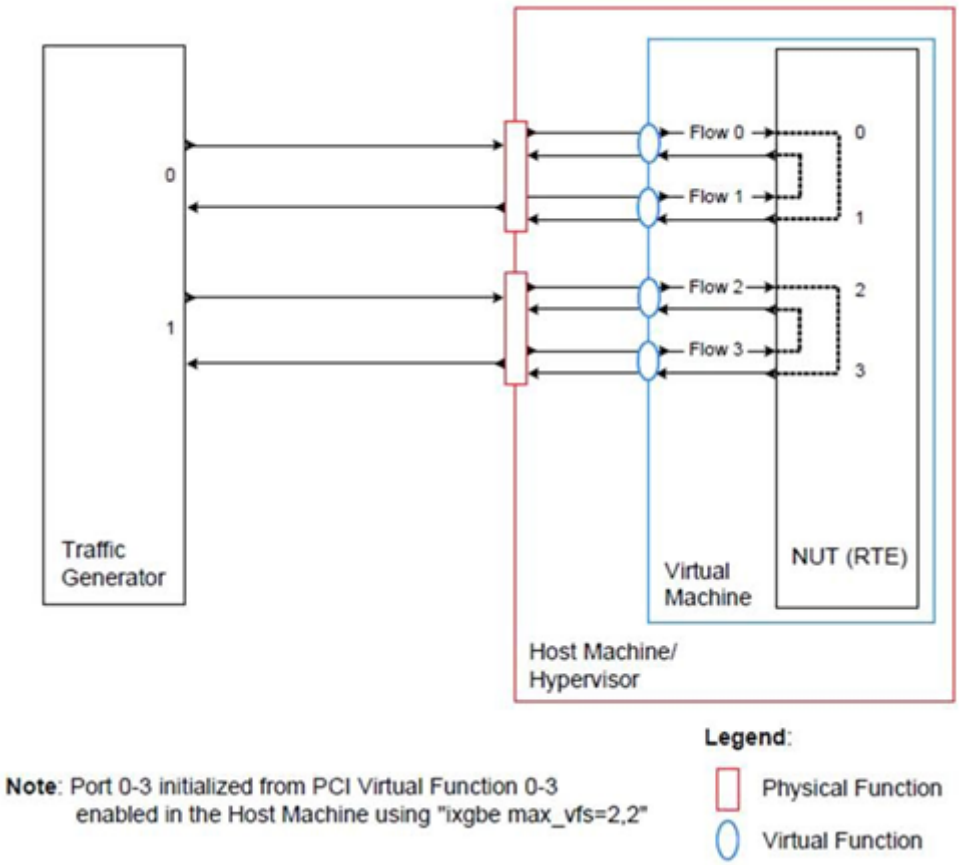


Fig. 4.6: Performance Benchmark Setup (Virtualized Environment)

Fig. 4.7: Virtual Machine to Virtual Machine communication.

Virtual Function Setup Instructions

This application can use the virtual function available in the system and therefore can be used in a virtual machine without passing through the whole Network Device into a guest machine in a virtualized scenario. The virtual functions can be enabled in the host machine or the hypervisor with the respective physical function driver.

For example, in a Linux* host machine, it is possible to enable a virtual function using the following command:

```
modprobe ixgbe max_vfs=2,2
```

This command enables two Virtual Functions on each of Physical Function of the NIC, with two physical ports in the PCI configuration space. It is important to note that enabled Virtual Function 0 and 2 would belong to Physical Function 0 and Virtual Function 1 and 3 would belong to Physical Function 1, in this case enabling a total of four Virtual Functions.

4.18.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd` sub-directory.

4.18.3 Running the Application

The application requires a number of command line options:

```
./build/l2fwd [EAL options] -- -p PORTMASK [-q NQ] --[no-]mac-updating
```

where,

- p PORTMASK: A hexadecimal bitmask of the ports to configure
- q NQ: A number of queues (=ports) per lcore (default is 1)
- –[no-]mac-updating: Enable or disable MAC addresses updating (enabled by default).

To run the application in linux environment with 4 lcores, 16 ports and 8 RX queues per lcore and MAC address updating enabled, issue the command:

```
$ ./build/l2fwd -l 0-3 -n 4 -- -q 8 -p ffff
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.18.4 Explanation

The following sections provide some explanation of the code.

Command Line Arguments

The L2 Forwarding sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments. The preferred way to parse parameters is to use the getopt() function, since it is part of a well-defined and portable library.

The parsing of arguments is done in the l2fwd_parse_args() function. The method of argument parsing is not described here. Refer to the *glibc getopt(3)* man page for details.

EAL arguments are parsed first, then application-specific arguments. This is done at the beginning of the main() function:

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");

argc -= ret;
argv += ret;

/* parse application arguments (after the EAL ones) */

ret = l2fwd_parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid L2FWD arguments\n");
```

Mbuf Pool Initialization

Once the arguments are parsed, the mbuf pool is created. The mbuf pool contains a set of mbuf objects that will be used by the driver and the application to store network packet data:

```
/* create the mbuf pool */

l2fwd_pktmbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", NB_MBUF,
    MEMPOOL_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE,
    rte_socket_id());

if (l2fwd_pktmbuf_pool == NULL)
    rte_panic("Cannot init mbuf pool\n");
```

The rte_mempool is a generic structure used to handle pools of objects. In this case, it is necessary to create a pool that will be used by the driver. The number of allocated pkt mbufs is NB_MBUF, with a data room size of RTE_MBUF_DEFAULT_BUF_SIZE each. A per-lcore cache of 32 mbufs is kept. The memory is allocated in NUMA socket 0, but it is possible to extend this code to allocate one mbuf pool per socket.

The rte_pktmbuf_pool_create() function uses the default mbuf pool and mbuf initializers, respectively rte_pktmbuf_pool_init() and rte_pktmbuf_init(). An advanced application may want to use the mempool API to create the mbuf pool with more control.

Driver Initialization

The main part of the code in the main() function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the Poll Mode Driver in the *DPDK Programmer's Guide* - Rel 1.4 EAR and the *DPDK API Reference*.

```
/* reset l2fwd_dst_ports */

for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++)
    l2fwd_dst_ports[portid] = 0;

last_port = 0;

/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */

RTE_ETH_FOREACH_DEV(portid) {
    /* skip ports that are not enabled */

    if ((l2fwd_enabled_port_mask & (1 << portid)) == 0)
       continue;

    if (nb_ports_in_mask % 2) {
        l2fwd_dst_ports[portid] = last_port;
        l2fwd_dst_ports[last_port] = portid;
    }
    else
       last_port = portid;

    nb_ports_in_mask++;

    rte_eth_dev_info_get((uint8_t) portid, &dev_info);
}
```

The next step is to configure the RX and TX queues. For each port, there is only one RX queue (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. The rte_eth_dev_configure() function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t)portid, 1, 1, &port_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot configure device: "
        "err=%d, port=%u\n",
        ret, portid);
```

RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the -q option, which specifies the number of queues per lcore.

For example, if the user specifies -q 4, the application is able to poll four ports with one lcore. If there are 16 ports on the target (and if the portmask argument is -p ffff), the application will need four lcores to poll all the ports.

```
ret = rte_eth_rx_queue_setup((uint8_t) portid, 0, nb_rxd, SOCKET0, &rx_conf, l2fwd_pktmbuf_
↪pool);
if (ret < 0)
```

(continues on next page)

(continued from previous page)

```
    rte_exit(EXIT_FAILURE, "rte_eth_rx_queue_setup: "
        "err=%d, port=%u\n",
        ret, portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called struct lcore_queue_conf.

```
struct lcore_queue_conf {
    unsigned n_rx_port;
    unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE];
    struct mbuf_table tx_mbufs[L2FWD_MAX_PORTS];
} rte_cache_aligned;

struct lcore_queue_conf lcore_queue_conf[RTE_MAX_LCORE];
```

The values n_rx_port and rx_port_list[] are used in the main packet processing loop (see *Receive, Process and Transmit Packets*).

TX Queue Initialization

Each lcore should be able to transmit on any port. For every port, a single TX queue is initialized.

```
/* init one TX queue on each port */

fflush(stdout);

ret = rte_eth_tx_queue_setup((uint8_t) portid, 0, nb_txd, rte_eth_dev_socket_id(portid), &tx_
↪conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup:err=%d, port=%u\n", ret, (unsigned) portid);
```

The global configuration for TX queues is stored in a static structure:

```
static const struct rte_eth_txconf tx_conf = {
    .tx_thresh = {
        .pthresh = TX_PTHRESH,
        .hthresh = TX_HTHRESH,
        .wthresh = TX_WTHRESH,
    },
    .tx_free_thresh = RTE_TEST_TX_DESC_DEFAULT + 1, /* disable feature */
};
```

Receive, Process and Transmit Packets

In the l2fwd_main_loop() function, the main task is to read ingress packets from the RX queues. This is done using the following code:

```
/*
 * Read packet from RX queues
 */

for (i = 0; i < qconf->n_rx_port; i++) {
    portid = qconf->rx_port_list[i];
    nb_rx = rte_eth_rx_burst((uint8_t) portid, 0,  pkts_burst, MAX_PKT_BURST);
```

(continues on next page)

(continued from previous page)

```
    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
        rte_prefetch0[rte_pktmbuf_mtod(m, void *)); l2fwd_simple_forward(m, portid);
    }
}
```

Packets are read in a burst of size MAX_PKT_BURST. The rte_eth_rx_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

Then, each mbuf in the table is processed by the l2fwd_simple_forward() function. The processing is very simple: process the TX port from the RX port, then replace the source and destination MAC addresses if MAC addresses updating is enabled.

Note: In the following code, one line for getting the output port requires some explanation.

During the initialization process, a static array of destination ports (l2fwd_dst_ports[]) is filled such that for each source port, a destination port is assigned that is either the next or previous enabled port from the portmask. Naturally, the number of ports in the portmask must be even, otherwise, the application exits.

```
static void
l2fwd_simple_forward(struct rte_mbuf *m, unsigned portid)
{
    struct rte_ether_hdr *eth;
    void *tmp;
    unsigned dst_port;

    dst_port = l2fwd_dst_ports[portid];

    eth = rte_pktmbuf_mtod(m, struct rte_ether_hdr *);

    /* 02:00:00:00:00:xx */

    tmp = &eth->d_addr.addr_bytes[0];

    *((uint64_t *)tmp) = 0x000000000002 + ((uint64_t) dst_port << 40);

    /* src addr */

    rte_ether_addr_copy(&l2fwd_ports_eth_addr[dst_port], &eth->s_addr);

    l2fwd_send_packet(m, (uint8_t) dst_port);
}
```

Then, the packet is sent using the l2fwd_send_packet (m, dst_port) function. For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the l2fwd_send_burst() function directly from the main loop to send all the received packets on the same TX port, using the burst-oriented send function, which is more efficient.

However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that, so the same approach can be reused in a more complex application.

The l2fwd_send_packet() function stores the packet in a per-lcore and per-txport table. If the table is full, the whole packets table is transmitted using the l2fwd_send_burst() function:

```
/* Send the packet on an output interface */

static int
l2fwd_send_packet(struct rte_mbuf *m, uint16_t port)
{
    unsigned lcore_id, len;
    struct lcore_queue_conf *qconf;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = m;
    len++;

    /* enough pkts to be sent */

    if (unlikely(len == MAX_PKT_BURST)) {
        l2fwd_send_burst(qconf, MAX_PKT_BURST, port);
        len = 0;
    }

    qconf->tx_mbufs[port].len = len; return 0;
}
```

To ensure that no packets remain in the tables, each lcore does a draining of TX queue in its main loop. This technique introduces some latency when there are not many packets to send, however it improves performance:

```
cur_tsc = rte_rdtsc();

/*
 *   TX burst queue drain
 */

diff_tsc = cur_tsc - prev_tsc;

if (unlikely(diff_tsc > drain_tsc)) {
    for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
        if (qconf->tx_mbufs[portid].len == 0)
            continue;

        l2fwd_send_burst(&lcore_queue_conf[lcore_id], qconf->tx_mbufs[portid].len, (uint8_t)␣
↪portid);

        qconf->tx_mbufs[portid].len = 0;
    }

    /* if timer is enabled */

    if (timer_period > 0) {
        /* advance the timer */

        timer_tsc += diff_tsc;

        /* if timer has reached its timeout */

        if (unlikely(timer_tsc >= (uint64_t) timer_period)) {
            /* do this only on master core */

            if (lcore_id == rte_get_master_lcore()) {
                print_stats();
```

(continues on next page)

(continued from previous page)

```
                /* reset the timer */
                timer_tsc = 0;
            }
        }
    }

    prev_tsc = cur_tsc;
}
```

4.19 L2 Forwarding Eventdev Sample Application

The L2 Forwarding eventdev sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) to demonstrate usage of poll and event mode packet I/O mechanism.

4.19.1 Overview

The L2 Forwarding eventdev sample application, performs L2 forwarding for each packet that is received on an RX_PORT. The destination port is the adjacent port from the enabled portmask, that is, if the first four ports are enabled (portmask=0x0f), ports 1 and 2 forward into each other, and ports 3 and 4 forward into each other. Also, if MAC addresses updating is enabled, the MAC addresses are affected as follows:

- The source MAC address is replaced by the TX_PORT MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

Application receives packets from RX_PORT using below mentioned methods:

- Poll mode
- Eventdev mode (default)

This application can be used to benchmark performance using a traffic-generator, as shown in the Fig. 4.8.

Fig. 4.8: Performance Benchmark Setup (Basic Environment)

4.19.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd-event` sub-directory.

4.19.3 Running the Application

The application requires a number of command line options:

```
./build/l2fwd-event [EAL options] -- -p PORTMASK [-q NQ] --[no-]mac-updating --mode=MODE --
↪eventq-sched=SCHED_MODE
```

where,

- p PORTMASK: A hexadecimal bitmask of the ports to configure
- q NQ: A number of queues (=ports) per lcore (default is 1)
- –[no-]mac-updating: Enable or disable MAC addresses updating (enabled by default).
- –mode=MODE: Packet transfer mode for I/O, poll or eventdev. Eventdev by default.
- –eventq-sched=SCHED_MODE: Event queue schedule mode, Ordered, Atomic or Parallel. Atomic by default.
- –config: Configure forwarding port pair mapping. Alternate port pairs by default.

Sample usage commands are given below to run the application into different mode:

Poll mode with 4 lcores, 16 ports and 8 RX queues per lcore and MAC address updating enabled, issue the command:

```
./build/l2fwd-event -l 0-3 -n 4 -- -q 8 -p ffff --mode=poll
```

Eventdev mode with 4 lcores, 16 ports , sched method ordered and MAC address updating enabled, issue the command:

```
./build/l2fwd-event -l 0-3 -n 4 -- -p ffff --eventq-sched=ordered
```

or

```
./build/l2fwd-event -l 0-3 -n 4 -- -q 8 -p ffff --mode=eventdev --eventq-sched=ordered
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

To run application with S/W scheduler, it uses following DPDK services:

- Software scheduler
- Rx adapter service function
- Tx adapter service function

Application needs service cores to run above mentioned services. Service cores must be provided as EAL parameters along with the –vdev=event_sw0 to enable S/W scheduler. Following is the sample command:

```
./build/l2fwd-event -l 0-7 -s 0-3 -n 4 --vdev event_sw0 -- -q 8 -p ffff --mode=eventdev --
↪eventq-sched=ordered
```

4.19.4 Explanation

The following sections provide some explanation of the code.

Command Line Arguments

The L2 Forwarding eventdev sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments. The preferred way to parse parameters is to use the getopt() function, since it is part of a well-defined and portable library.

The parsing of arguments is done in the **l2fwd_parse_args()** function for non eventdev parameters and in **parse_eventdev_args()** for eventdev parameters. The method of argument parsing is not described here. Refer to the *glibc getopt(3)* man page for details.

EAL arguments are parsed first, then application-specific arguments. This is done at the beginning of the main() function and eventdev parameters are parsed in eventdev_resource_setup() function during eventdev setup:

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_panic("Invalid EAL arguments\n");

argc -= ret;
argv += ret;

/* parse application arguments (after the EAL ones) */

ret = l2fwd_parse_args(argc, argv);
if (ret < 0)
    rte_panic("Invalid L2FWD arguments\n");
.
.
.

/* Parse eventdev command line options */
ret = parse_eventdev_args(argc, argv);
if (ret < 0)
    return ret;
```

Mbuf Pool Initialization

Once the arguments are parsed, the mbuf pool is created. The mbuf pool contains a set of mbuf objects that will be used by the driver and the application to store network packet data:

```
/* create the mbuf pool */

l2fwd_pktmbuf_pool = rte_pktmbuf_pool_create("mbuf_pool", NB_MBUF,
                                             MEMPOOL_CACHE_SIZE, 0,
                                             RTE_MBUF_DEFAULT_BUF_SIZE,
                                             rte_socket_id());
if (l2fwd_pktmbuf_pool == NULL)
    rte_panic("Cannot init mbuf pool\n");
```

The rte_mempool is a generic structure used to handle pools of objects. In this case, it is necessary to create a pool that will be used by the driver. The number of allocated pkt mbufs is NB_MBUF, with

a data room size of RTE_MBUF_DEFAULT_BUF_SIZE each. A per-lcore cache of 32 mbufs is kept. The memory is allocated in NUMA socket 0, but it is possible to extend this code to allocate one mbuf pool per socket.

The rte_pktmbuf_pool_create() function uses the default mbuf pool and mbuf initializers, respectively rte_pktmbuf_pool_init() and rte_pktmbuf_init(). An advanced application may want to use the mempool API to create the mbuf pool with more control.

Driver Initialization

The main part of the code in the main() function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the Poll Mode and Event mode Driver in the *DPDK Programmer's Guide* - Rel 1.4 EAR and the *DPDK API Reference*.

```
/* reset l2fwd_dst_ports */

for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++)
    l2fwd_dst_ports[portid] = 0;

last_port = 0;

/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */

RTE_ETH_FOREACH_DEV(portid) {
    /* skip ports that are not enabled */

    if ((l2fwd_enabled_port_mask & (1 << portid)) == 0)
       continue;

    if (nb_ports_in_mask % 2) {
        l2fwd_dst_ports[portid] = last_port;
        l2fwd_dst_ports[last_port] = portid;
    }
    else
       last_port = portid;

    nb_ports_in_mask++;

    rte_eth_dev_info_get((uint8_t) portid, &dev_info);
}
```

The next step is to configure the RX and TX queues. For each port, there is only one RX queue (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. The rte_eth_dev_configure() function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t)portid, 1, 1, &port_conf);
if (ret < 0)
    rte_panic("Cannot configure device: err=%d, port=%u\n",
              ret, portid);
```

RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the -q option, which specifies the number of queues per lcore.

For example, if the user specifies -q 4, the application is able to poll four ports with one lcore. If there are 16 ports on the target (and if the portmask argument is -p ffff), the application will need four lcores to poll all the ports.

```
ret = rte_eth_rx_queue_setup((uint8_t) portid, 0, nb_rxd, SOCKET0,
                             &rx_conf, l2fwd_pktmbuf_pool);
if (ret < 0)

    rte_panic("rte_eth_rx_queue_setup: err=%d, port=%u\n",
              ret, portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called struct lcore_queue_conf.

```
struct lcore_queue_conf {
    unsigned n_rx_port;
    unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE];
    struct mbuf_table tx_mbufs[L2FWD_MAX_PORTS];
} rte_cache_aligned;

struct lcore_queue_conf lcore_queue_conf[RTE_MAX_LCORE];
```

The values n_rx_port and rx_port_list[] are used in the main packet processing loop (see *Receive, Process and Transmit Packets*).

TX Queue Initialization

Each lcore should be able to transmit on any port. For every port, a single TX queue is initialized.

```
/* init one TX queue on each port */

fflush(stdout);

ret = rte_eth_tx_queue_setup((uint8_t) portid, 0, nb_txd,
                             rte_eth_dev_socket_id(portid), &tx_conf);
if (ret < 0)
    rte_panic("rte_eth_tx_queue_setup:err=%d, port=%u\n",
              ret, (unsigned) portid);
```

To configure eventdev support, application setups following components:

- Event dev
- Event queue
- Event Port
- Rx/Tx adapters
- Ethernet ports

Event device Initialization

Application can use either H/W or S/W based event device scheduler implementation and supports single instance of event device. It configures event device as per below configuration

```
struct rte_event_dev_config event_d_conf = {
     .nb_event_queues = ethdev_count, /* Dedicated to each Ethernet port */
     .nb_event_ports = num_workers, /* Dedicated to each lcore */
     .nb_events_limit  = 4096,
     .nb_event_queue_flows = 1024,
     .nb_event_port_dequeue_depth = 128,
     .nb_event_port_enqueue_depth = 128
};

ret = rte_event_dev_configure(event_d_id, &event_d_conf);
if (ret < 0)
     rte_panic("Error in configuring event device\n");
```

In case of S/W scheduler, application runs eventdev scheduler service on service core. Application retrieves service id and finds the best possible service core to run S/W scheduler.

```
rte_event_dev_info_get(evt_rsrc->event_d_id, &evdev_info);
if (evdev_info.event_dev_cap  & RTE_EVENT_DEV_CAP_DISTRIBUTED_SCHED) {
        ret = rte_event_dev_service_id_get(evt_rsrc->event_d_id,
                        &service_id);
        if (ret != -ESRCH && ret != 0)
                rte_panic("Error in starting eventdev service\n");
        l2fwd_event_service_enable(service_id);
}
```

Event queue Initialization

Each Ethernet device is assigned a dedicated event queue which will be linked to all available event ports i.e. each lcore can dequeue packets from any of the Ethernet ports.

```
struct rte_event_queue_conf event_q_conf = {
     .nb_atomic_flows = 1024,
     .nb_atomic_order_sequences = 1024,
     .event_queue_cfg = 0,
     .schedule_type = RTE_SCHED_TYPE_ATOMIC,
     .priority = RTE_EVENT_DEV_PRIORITY_HIGHEST
};

/* User requested sched mode */
event_q_conf.schedule_type = eventq_sched_mode;
for (event_q_id = 0; event_q_id < ethdev_count; event_q_id++) {
     ret = rte_event_queue_setup(event_d_id, event_q_id,
                                         &event_q_conf);
     if (ret < 0)
           rte_panic("Error in configuring event queue\n");
}
```

In case of S/W scheduler, an extra event queue is created which will be used for Tx adapter service function for enqueue operation.

Event port Initialization

Each worker thread is assigned a dedicated event port for enq/deq operations to/from an event device. All event ports are linked with all available event queues.

```
struct rte_event_port_conf event_p_conf = {
     .dequeue_depth = 32,
     .enqueue_depth = 32,
     .new_event_threshold = 4096
};

for (event_p_id = 0; event_p_id < num_workers; event_p_id++) {
     ret = rte_event_port_setup(event_d_id, event_p_id,
                                &event_p_conf);
     if (ret < 0)
           rte_panic("Error in configuring event port %d\n", event_p_id);

     ret = rte_event_port_link(event_d_id, event_p_id, NULL,
                               NULL, 0);
     if (ret < 0)
           rte_panic("Error in linking event port %d to queue\n",
                     event_p_id);
}
```

In case of S/W scheduler, an extra event port is created by DPDK library which is retrieved by the application and same will be used by Tx adapter service.

```
ret = rte_event_eth_tx_adapter_event_port_get(tx_adptr_id, &tx_port_id);
if (ret)
        rte_panic("Failed to get Tx adapter port id: %d\n", ret);

ret = rte_event_port_link(event_d_id, tx_port_id,
                          &evt_rsrc.evq.event_q_id[
                                evt_rsrc.evq.nb_queues - 1],
                          NULL, 1);
if (ret != 1)
        rte_panic("Unable to link Tx adapter port to Tx queue:err=%d\n",
                  ret);
```

Rx/Tx adapter Initialization

Each Ethernet port is assigned a dedicated Rx/Tx adapter for H/W scheduler. Each Ethernet port's Rx queues are connected to its respective event queue at priority 0 via Rx adapter configuration and Ethernet port's tx queues are connected via Tx adapter.

```
RTE_ETH_FOREACH_DEV(port_id) {
        if ((rsrc->enabled_port_mask & (1 << port_id)) == 0)
                continue;
        ret = rte_event_eth_rx_adapter_create(adapter_id, event_d_id,
                                              &evt_rsrc->def_p_conf);
        if (ret)
                rte_panic("Failed to create rx adapter[%d]\n",
                          adapter_id);

        /* Configure user requested sched type*/
        eth_q_conf.ev.sched_type = rsrc->sched_type;
        eth_q_conf.ev.queue_id = evt_rsrc->evq.event_q_id[q_id];
        ret = rte_event_eth_rx_adapter_queue_add(adapter_id, port_id,
```

(continues on next page)

(continued from previous page)

```
                                                -1, &eth_q_conf);
        if (ret)
                rte_panic("Failed to add queues to Rx adapter\n");

        ret = rte_event_eth_rx_adapter_start(adapter_id);
        if (ret)
                rte_panic("Rx adapter[%d] start Failed\n", adapter_id);

        evt_rsrc->rx_adptr.rx_adptr[adapter_id] = adapter_id;
        adapter_id++;
        if (q_id < evt_rsrc->evq.nb_queues)
                q_id++;
}

adapter_id = 0;
RTE_ETH_FOREACH_DEV(port_id) {
        if ((rsrc->enabled_port_mask & (1 << port_id)) == 0)
                continue;
        ret = rte_event_eth_tx_adapter_create(adapter_id, event_d_id,
                                        &evt_rsrc->def_p_conf);
        if (ret)
                rte_panic("Failed to create tx adapter[%d]\n",
                          adapter_id);

        ret = rte_event_eth_tx_adapter_queue_add(adapter_id, port_id,
                                                -1);
        if (ret)
                rte_panic("Failed to add queues to Tx adapter\n");

        ret = rte_event_eth_tx_adapter_start(adapter_id);
        if (ret)
                rte_panic("Tx adapter[%d] start Failed\n", adapter_id);

        evt_rsrc->tx_adptr.tx_adptr[adapter_id] = adapter_id;
        adapter_id++;
}
```

For S/W scheduler instead of dedicated adapters, common Rx/Tx adapters are configured which will be shared among all the Ethernet ports. Also DPDK library need service cores to run internal services for Rx/Tx adapters. Application gets service id for Rx/Tx adapters and after successful setup it runs the services on dedicated service cores.

```
for (i = 0; i < evt_rsrc->rx_adptr.nb_rx_adptr; i++) {
        ret = rte_event_eth_rx_adapter_caps_get(evt_rsrc->event_d_id,
                        evt_rsrc->rx_adptr.rx_adptr[i], &caps);
        if (ret < 0)
                rte_panic("Failed to get Rx adapter[%d] caps\n",
                          evt_rsrc->rx_adptr.rx_adptr[i]);
        ret = rte_event_eth_rx_adapter_service_id_get(
                                        evt_rsrc->event_d_id,
                                        &service_id);
        if (ret != -ESRCH && ret != 0)
                rte_panic("Error in starting Rx adapter[%d] service\n",
                          evt_rsrc->rx_adptr.rx_adptr[i]);
        l2fwd_event_service_enable(service_id);
}

for (i = 0; i < evt_rsrc->tx_adptr.nb_tx_adptr; i++) {
        ret = rte_event_eth_tx_adapter_caps_get(evt_rsrc->event_d_id,
                        evt_rsrc->tx_adptr.tx_adptr[i], &caps);
```

(continues on next page)

(continued from previous page)

```
        if (ret < 0)
                rte_panic("Failed to get Rx adapter[%d] caps\n",
                          evt_rsrc->tx_adptr.tx_adptr[i]);
        ret = rte_event_eth_tx_adapter_service_id_get(
                        evt_rsrc->event_d_id,
                        &service_id);
        if (ret != -ESRCH && ret != 0)
                rte_panic("Error in starting Rx adapter[%d] service\n",
                          evt_rsrc->tx_adptr.tx_adptr[i]);
        l2fwd_event_service_enable(service_id);
}
```

Receive, Process and Transmit Packets

In the **l2fwd_main_loop()** function, the main task is to read ingress packets from the RX queues. This is done using the following code:

```
/*
 * Read packet from RX queues
 */

for (i = 0; i < qconf->n_rx_port; i++) {
    portid = qconf->rx_port_list[i];
    nb_rx = rte_eth_rx_burst((uint8_t) portid, 0,  pkts_burst,
                             MAX_PKT_BURST);

    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
        rte_prefetch0(rte_pktmbuf_mtod(m, void *));
        l2fwd_simple_forward(m, portid);
    }
}
```

Packets are read in a burst of size MAX_PKT_BURST. The rte_eth_rx_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

Then, each mbuf in the table is processed by the l2fwd_simple_forward() function. The processing is very simple: process the TX port from the RX port, then replace the source and destination MAC addresses if MAC addresses updating is enabled.

During the initialization process, a static array of destination ports (l2fwd_dst_ports[]) is filled such that for each source port, a destination port is assigned that is either the next or previous enabled port from the portmask. If number of ports are odd in portmask then packet from last port will be forwarded to first port i.e. if portmask=0x07, then forwarding will take place like p0—>p1, p1—>p2, p2—>p0.

Also to optimize enqueue operation, l2fwd_simple_forward() stores incoming mbufs up to MAX_PKT_BURST. Once it reaches up to limit, all packets are transmitted to destination ports.

```
static void
l2fwd_simple_forward(struct rte_mbuf *m, uint32_t portid)
{
    uint32_t dst_port;
    int32_t sent;
    struct rte_eth_dev_tx_buffer *buffer;

    dst_port = l2fwd_dst_ports[portid];
```

(continues on next page)

(continued from previous page)

```
    if (mac_updating)
        l2fwd_mac_updating(m, dst_port);

    buffer = tx_buffer[dst_port];
    sent = rte_eth_tx_buffer(dst_port, 0, buffer, m);
    if (sent)
    port_statistics[dst_port].tx += sent;
}
```

For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the rte_eth_tx_buffer() function directly from the main loop to send all the received packets on the same TX port, using the burst-oriented send function, which is more efficient.

However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that, so the same approach can be reused in a more complex application.

To ensure that no packets remain in the tables, each lcore does a draining of TX queue in its main loop. This technique introduces some latency when there are not many packets to send, however it improves performance:

```
cur_tsc = rte_rdtsc();

/*
* TX burst queue drain
*/
diff_tsc = cur_tsc - prev_tsc;
if (unlikely(diff_tsc > drain_tsc)) {
        for (i = 0; i < qconf->n_rx_port; i++) {
                portid = l2fwd_dst_ports[qconf->rx_port_list[i]];
                buffer = tx_buffer[portid];
                sent = rte_eth_tx_buffer_flush(portid, 0,
                                               buffer);
                if (sent)
                        port_statistics[portid].tx += sent;
        }

        /* if timer is enabled */
        if (timer_period > 0) {
                /* advance the timer */
                timer_tsc += diff_tsc;

                /* if timer has reached its timeout */
                if (unlikely(timer_tsc >= timer_period)) {
                        /* do this only on master core */
                        if (lcore_id == rte_get_master_lcore()) {
                                print_stats();
                                /* reset the timer */
                                timer_tsc = 0;
                        }
                }
        }

        prev_tsc = cur_tsc;
}
```

In the **l2fwd_event_loop()** function, the main task is to read ingress packets from the event ports. This is done using the following code:

```
/* Read packet from eventdev */
nb_rx = rte_event_dequeue_burst(event_d_id, event_p_id,
                                events, deq_len, 0);
if (nb_rx == 0) {
        rte_pause();
        continue;
}

for (i = 0; i < nb_rx; i++) {
        mbuf[i] = events[i].mbuf;
        rte_prefetch0(rte_pktmbuf_mtod(mbuf[i], void *));
}
```

Before reading packets, deq_len is fetched to ensure correct allowed deq length by the eventdev. The rte_event_dequeue_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

Then, each mbuf in the table is processed by the l2fwd_eventdev_forward() function. The processing is very simple: process the TX port from the RX port, then replace the source and destination MAC addresses if MAC addresses updating is enabled.

During the initialization process, a static array of destination ports (l2fwd_dst_ports[]) is filled such that for each source port, a destination port is assigned that is either the next or previous enabled port from the portmask. If number of ports are odd in portmask then packet from last port will be forwarded to first port i.e. if portmask=0x07, then forwarding will take place like p0—>p1, p1—>p2, p2—>p0.

l2fwd_eventdev_forward() does not stores incoming mbufs. Packet will forwarded be to destination ports via Tx adapter or generic event dev enqueue API depending H/W or S/W scheduler is used.

```
nb_tx = rte_event_eth_tx_adapter_enqueue(event_d_id, port_id, ev,
                                         nb_rx);
while (nb_tx < nb_rx && !rsrc->force_quit)
        nb_tx += rte_event_eth_tx_adapter_enqueue(
                        event_d_id, port_id,
                        ev + nb_tx, nb_rx - nb_tx);
```

4.20 L2 Forwarding Sample Application with Cache Allocation Technology (CAT)

Basic Forwarding sample application is a simple *skeleton* example of a forwarding application. It has been extended to make use of CAT via extended command line options and linking against the libpqos library.

It is intended as a demonstration of the basic components of a DPDK forwarding application and use of the libpqos library to program CAT. For more detailed implementations see the L2 and L3 forwarding sample applications.

CAT and Code Data Prioritization (CDP) features allow management of the CPU's last level cache. CAT introduces classes of service (COS) that are essentially bitmasks. In current CAT implementations, a bit in a COS bitmask corresponds to one cache way in last level cache. A CPU core is always assigned to one of the CAT classes. By programming CPU core assignment and COS bitmasks, applications can be given exclusive, shared, or mixed access to the CPU's last level cache. CDP extends CAT so that there are two bitmasks per COS, one for data and one for code. The number of classes and number of valid bits in a COS bitmask is CPU model specific and COS bitmasks need to be contiguous. Sample code

calls this bitmask `cbm` or capacity bitmask. By default, after reset, all CPU cores are assigned to COS 0 and all classes are programmed to allow fill into all cache ways. CDP is off by default.

For more information about CAT please see:

- https://github.com/01org/intel-cmt-cat

White paper demonstrating example use case:

- Increasing Platform Determinism with Platform Quality of Service for the Data Plane Development Kit

4.20.1 Compiling the Application

Note: Requires `libpqos` from Intel's intel-cmt-cat software package hosted on GitHub repository. For installation notes, please see `README` file.

GIT:

- https://github.com/01org/intel-cmt-cat

---

1. To compile the application export the path to PQoS lib and the DPDK source tree and go to the example directory:

```
export PQOS_INSTALL_PATH=/path/to/libpqos
```

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l2fwd-cat` sub-directory.

4.20.2 Running the Application

To run the example in a `linux` environment and enable CAT on cpus 0-2:

```
./build/l2fwd-cat -l 1 -n 4 -- --l3ca="0x3@(0-2)"
```

or to enable CAT and CDP on cpus 1,3:

```
./build/l2fwd-cat -l 1 -n 4 -- --l3ca="(0x00C00,0x00300)@(1,3)"
```

If CDP is not supported it will fail with following error message:

```
PQOS: CDP requested but not supported.
PQOS: Requested CAT configuration is not valid!
PQOS: Shutting down PQoS library...
EAL: Error - exiting with code: 1
  Cause: PQOS: L3CA init failed!
```

The option to enable CAT is:

- `--l3ca='<common_cbm@cpus>[,<(code_cbm,data_cbm)@cpus>...]'`:

  where `cbm` stands for capacity bitmask and must be expressed in hexadecimal form.

  `common_cbm` is a single mask, for a CDP enabled system, a group of two masks (`code_cbm` and `data_cbm`) is used.

(and) are necessary if it's a group.

cpus could be a single digit/range or a group and must be expressed in decimal form.

(and) are necessary if it's a group.

e.g. `--l3ca='0x00F00@(1,3),0x0FF00@(4-6),0xF0000@7'`

- cpus 1 and 3 share its 4 ways with cpus 4, 5 and 6;
- cpus 4, 5 and 6 share half (4 out of 8 ways) of its L3 with cpus 1 and 3;
- cpus 4, 5 and 6 have exclusive access to 4 out of 8 ways;
- cpu 7 has exclusive access to all of its 4 ways;

e.g. `--l3ca='(0x00C00,0x00300)@(1,3)'` for CDP enabled system

- cpus 1 and 3 have access to 2 ways for code and 2 ways for data, code and data ways are not overlapping.

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

To reset or list CAT configuration and control CDP please use pqos tool from Intel's intel-cmt-cat software package.

To enabled or disable CDP:

```
sudo ./pqos -S cdp-on

sudo ./pqos -S cdp-off
```

to reset CAT configuration:

```
sudo ./pqos -R
```

to list CAT config:

```
sudo ./pqos -s
```

For more info about pqos tool please see its man page or intel-cmt-cat wiki.

4.20.3 Explanation

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with rte_ and are explained in detail in the *DPDK API Documentation*.

The Main Function

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

The next task is to initialize the PQoS library and configure CAT. The `argc` and `argv` arguments are provided to the `cat_init()` function. The value returned is the number of parsed arguments:

```
int ret = cat_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "PQOS: L3CA init failed!\n");
```

`cat_init()` is a wrapper function which parses the command, validates the requested parameters and configures CAT accordingly.

Parsing of command line arguments is done in `parse_args(...)`. libpqos is then initialized with the `pqos_init(...)` call. Next, libpqos is queried for system CPU information and L3CA capabilities via `pqos_cap_get(...)` and `pqos_cap_get_type(..., PQOS_CAP_TYPE_L3CA, ...)` calls. When all capability and topology information is collected, the requested CAT configuration is validated. A check is then performed (on per socket basis) for a sufficient number of un-associated COS. COS are selected and configured via the `pqos_l3ca_set(...)` call. Finally, COS are associated to relevant CPUs via `pqos_l3ca_assoc_set(...)` calls.

`atexit(...)` is used to register `cat_exit(...)` to be called on a clean exit. `cat_exit(...)` performs a simple CAT clean-up, by associating COS 0 to all involved CPUs via `pqos_l3ca_assoc_set(...)` calls.

4.21 L3 Forwarding Sample Application

The L3 Forwarding application is a simple example of packet processing using DPDK to demonstrate usage of poll and event mode packet I/O mechanism. The application performs L3 forwarding.

4.21.1 Overview

The application demonstrates the use of the hash and LPM libraries in the DPDK to implement packet forwarding using poll or event mode PMDs for packet I/O. The initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)* and *L2 Forwarding Eventdev Sample Application*. The main difference from the L2 Forwarding sample application is that optionally packet can be Rx/Tx from/to eventdev instead of port directly and forwarding decision is made based on information read from the input packet.

Eventdev can optionally use S/W or H/W (if supported by platform) scheduler implementation for packet I/O based on run time parameters.

The lookup method is either hash-based or LPM-based and is selected at run time. When the selected lookup method is hash-based, a hash object is used to emulate the flow classification stage. The hash object is used in correlation with a flow table to map each input packet to its flow at runtime.

The hash lookup key is represented by a DiffServ 5-tuple composed of the following fields read from the input packet: Source IP Address, Destination IP Address, Protocol, Source Port and Destination Port. The ID of the output interface for the input packet is read from the identified flow table entry. The set of flows used by the application is statically configured and loaded into the hash at initialization time. When the selected lookup method is LPM based, an LPM object is used to emulate the forwarding stage for IPv4 packets. The LPM object is used as the routing table to identify the next hop for each input packet at runtime.

The LPM lookup key is represented by the Destination IP Address field read from the input packet. The ID of the output interface for the input packet is the next hop returned by the LPM lookup. The set of LPM rules used by the application is statically configured and loaded into the LPM object at initialization time.

In the sample application, hash-based forwarding supports IPv4 and IPv6. LPM-based forwarding supports IPv4 only.

4.21.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l3fwd` sub-directory.

4.21.3 Running the Application

The application has a number of command line options:

```
./l3fwd [EAL options] -- -p PORTMASK
                         [-P]
                         [-E]
                         [-L]
                         --config(port,queue,lcore)[,(port,queue,lcore)]
                         [--eth-dest=X,MM:MM:MM:MM:MM:MM]
                         [--enable-jumbo [--max-pkt-len PKTLEN]]
                         [--no-numa]
                         [--hash-entry-num]
                         [--ipv6]
                         [--parse-ptype]
                         [--per-port-pool]
                         [--mode]
                         [--eventq-sched]
                         [--event-eth-rxqs]
```

Where,

- `-p PORTMASK:` Hexadecimal bitmask of ports to configure
- `-P:` Optional, sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- `-E:` Optional, enable exact match.
- `-L:` Optional, enable longest prefix match.
- `--config (port,queue,lcore)[,(port,queue,lcore)]:` Determines which queues from which ports are mapped to which cores.
- `--eth-dest=X,MM:MM:MM:MM:MM:MM:` Optional, ethernet destination for port X.

- `--enable-jumbo:` Optional, enables jumbo frames.
- `--max-pkt-len:` Optional, under the premise of enabling jumbo, maximum packet length in decimal (64-9600).
- `--no-numa:` Optional, disables numa awareness.
- `--hash-entry-num:` Optional, specifies the hash entry number in hexadecimal to be setup.
- `--ipv6:` Optional, set if running ipv6 packets.
- `--parse-ptype:` Optional, set to use software to analyze packet type. Without this option, hardware will check the packet type.
- `--per-port-pool:` Optional, set to use independent buffer pools per port. Without this option, single buffer pool is used for all ports.
- `--mode:` Optional, Packet transfer mode for I/O, poll or eventdev.
- `--eventq-sched:` Optional, Event queue synchronization method, Ordered, Atomic or Parallel. Only valid if –mode=eventdev.
- `--event-eth-rxqs:` Optional, Number of ethernet RX queues per device. Only valid if –mode=eventdev.

For example, consider a dual processor socket platform with 8 physical cores, where cores 0-7 and 16-23 appear on socket 0, while cores 8-15 and 24-31 appear on socket 1.

To enable L3 forwarding between two ports, assuming that both ports are in the same socket, using two cores, cores 1 and 2, (which are in the same socket too), use the following command:

```
./build/l3fwd -l 1,2 -n 4 -- -p 0x3 --config="(0,0,1),(1,0,2)"
```

In this command:

- The -l option enables cores 1, 2
- The -p option enables ports 0 and 1
- The –config option enables one queue on each port and maps each (port,queue) pair to a specific core. The following table shows the mapping in this example:

Port	Queue	lcore	Description
0	0	1	Map queue 0 from port 0 to lcore 1.
1	0	2	Map queue 0 from port 1 to lcore 2.

To use eventdev mode with sync method **ordered** on above mentioned environment, Following is the sample command:

```
./build/l3fwd -l 0-3 -n 4 -w <event device> -- -p 0x3 --eventq-sched=ordered
```

or

```
./build/l3fwd -l 0-3 -n 4 -w <event device> -- -p 0x03 --mode=eventdev --eventq-sched=ordered
```

In this command:

- -w option whitelist the event device supported by platform. Way to pass this device may vary based on platform.

- The –mode option defines PMD to be used for packet I/O.
- The –eventq-sched option enables synchronization menthod of event queue so that packets will be scheduled accordingly.

If application uses S/W scheduler, it uses following DPDK services:

- Software scheduler
- Rx adapter service function
- Tx adapter service function

Application needs service cores to run above mentioned services. Service cores must be provided as EAL parameters along with the –vdev=event_sw0 to enable S/W scheduler. Following is the sample command:

```
./build/l3fwd -l 0-7 -s 0xf0000 -n 4 --vdev event_sw0 -- -p 0x3 --mode=eventdev --eventq-
↪sched=ordered
```

In case of eventdev mode, *–config* option is not used for ethernet port configuration. Instead each ethernet port will be configured with mentioned setup:

- Single Rx/Tx queue
- Each Rx queue will be connected to event queue via Rx adapter.
- Each Tx queue will be connected via Tx adapter.

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.21.4 Explanation

The following sections provide some explanation of the sample application code. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the *L2 Forwarding Sample Application (in Real and Virtualized Environments)* and *L2 Forwarding Eventdev Sample Application*. The following sections describe aspects that are specific to the L3 Forwarding sample application.

Hash Initialization

The hash object is created and loaded with the pre-configured entries read from a global array, and then generate the expected 5-tuple as key to keep consistence with those of real flow for the convenience to execute hash performance test on 4M/8M/16M flows.

Note: The Hash initialization will setup both ipv4 and ipv6 hash table, and populate the either table depending on the value of variable ipv6. To support the hash performance test with up to 8M single direction flows/16M bi-direction flows, populate_ipv4_many_flow_into_table() function will populate the hash table with specified hash table entry number(default 4M).

Note: Value of global variable ipv6 can be specified with –ipv6 in the command line. Value of global variable hash_entry_number, which is used to specify the total hash entry number for all used ports in

hash performance test, can be specified with –hash-entry-num VALUE in command line, being its default value 4.

```
#if (APP_LOOKUP_METHOD == APP_LOOKUP_EXACT_MATCH)

    static void
    setup_hash(int socketid)
    {
        // ...

        if (hash_entry_number != HASH_ENTRY_NUMBER_DEFAULT) {
            if (ipv6 == 0) {
                /* populate the ipv4 hash */
                populate_ipv4_many_flow_into_table(ipv4_l3fwd_lookup_struct[socketid], hash_
↪entry_number);
            } else {
                /* populate the ipv6 hash */
                populate_ipv6_many_flow_into_table( ipv6_l3fwd_lookup_struct[socketid], hash_
↪entry_number);
            }
        } else
            if (ipv6 == 0) {
                /* populate the ipv4 hash */
                populate_ipv4_few_flow_into_table(ipv4_l3fwd_lookup_struct[socketid]);
            } else {
                /* populate the ipv6 hash */
                populate_ipv6_few_flow_into_table(ipv6_l3fwd_lookup_struct[socketid]);
            }
        }
    }
#endif
```

LPM Initialization

The LPM object is created and loaded with the pre-configured entries read from a global array.

```
#if (APP_LOOKUP_METHOD == APP_LOOKUP_LPM)

static void
setup_lpm(int socketid)
{
    unsigned i;
    int ret;
    char s[64];

    /* create the LPM table */

    snprintf(s, sizeof(s), "IPV4_L3FWD_LPM_%d", socketid);

    ipv4_l3fwd_lookup_struct[socketid] = rte_lpm_create(s, socketid, IPV4_L3FWD_LPM_MAX_RULES,␣
↪0);

    if (ipv4_l3fwd_lookup_struct[socketid] == NULL)
        rte_exit(EXIT_FAILURE, "Unable to create the l3fwd LPM table"
            " on socket %d\n", socketid);

    /* populate the LPM table */

    for (i = 0; i < IPV4_L3FWD_NUM_ROUTES; i++) {
```

(continues on next page)

(continued from previous page)

```
        /* skip unused ports */

        if ((1 << ipv4_l3fwd_route_array[i].if_out & enabled_port_mask) == 0)
            continue;

        ret = rte_lpm_add(ipv4_l3fwd_lookup_struct[socketid], ipv4_l3fwd_route_array[i].ip,
                                ipv4_l3fwd_route_array[i].depth, ipv4_l3fwd_route_array[i].if_
↪out);

        if (ret < 0) {
            rte_exit(EXIT_FAILURE, "Unable to add entry %u to the "
                    "l3fwd LPM table on socket %d\n", i, socketid);
        }

        printf("LPM: Adding route 0x%08x / %d (%d)\n",
            (unsigned)ipv4_l3fwd_route_array[i].ip, ipv4_l3fwd_route_array[i].depth, ipv4_
↪l3fwd_route_array[i].if_out);
    }
}
#endif
```

Packet Forwarding for Hash-based Lookups

For each input packet, the packet forwarding operation is done by the l3fwd_simple_forward() or simple_ipv4_fwd_4pkts() function for IPv4 packets or the simple_ipv6_fwd_4pkts() function for IPv6 packets. The l3fwd_simple_forward() function provides the basic functionality for both IPv4 and IPv6 packet forwarding for any number of burst packets received, and the packet forwarding decision (that is, the identification of the output interface for the packet) for hash-based lookups is done by the get_ipv4_dst_port() or get_ipv6_dst_port() function. The get_ipv4_dst_port() function is shown below:

```
static inline uint8_t
get_ipv4_dst_port(void *ipv4_hdr, uint16_t portid, lookup_struct_t *ipv4_l3fwd_lookup_struct)
{
    int ret = 0;
    union ipv4_5tuple_host key;

    ipv4_hdr = (uint8_t *)ipv4_hdr + offsetof(struct rte_ipv4_hdr, time_to_live);

    m128i data = _mm_loadu_si128(( m128i*)(ipv4_hdr));

    /* Get 5 tuple: dst port, src port, dst IP address, src IP address and protocol */

    key.xmm = _mm_and_si128(data, mask0);

    /* Find destination port */

    ret = rte_hash_lookup(ipv4_l3fwd_lookup_struct, (const void *)&key);

    return (uint8_t)((ret < 0)? portid : ipv4_l3fwd_out_if[ret]);
}
```

The get_ipv6_dst_port() function is similar to the get_ipv4_dst_port() function.

The simple_ipv4_fwd_4pkts() and simple_ipv6_fwd_4pkts() function are optimized for continuous 4 valid ipv4 and ipv6 packets, they leverage the multiple buffer optimization to boost the performance of forwarding packets with the exact match on hash table. The key code snippet of simple_ipv4_fwd_4pkts() is shown below:

```
static inline void
simple_ipv4_fwd_4pkts(struct rte_mbuf* m[4], uint16_t portid, struct lcore_conf *qconf)
{
    // ...

    data[0] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[0], unsigned char *) +␣
↪sizeof(struct rte_ether_hdr) + offsetof(struct rte_ipv4_hdr, time_to_live)));
    data[1] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[1], unsigned char *) +␣
↪sizeof(struct rte_ether_hdr) + offsetof(struct rte_ipv4_hdr, time_to_live)));
    data[2] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[2], unsigned char *) +␣
↪sizeof(struct rte_ether_hdr) + offsetof(struct rte_ipv4_hdr, time_to_live)));
    data[3] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[3], unsigned char *) +␣
↪sizeof(struct rte_ether_hdr) + offsetof(struct rte_ipv4_hdr, time_to_live)));

    key[0].xmm = _mm_and_si128(data[0], mask0);
    key[1].xmm = _mm_and_si128(data[1], mask0);
    key[2].xmm = _mm_and_si128(data[2], mask0);
    key[3].xmm = _mm_and_si128(data[3], mask0);

    const void *key_array[4] = {&key[0], &key[1], &key[2],&key[3]};

    rte_hash_lookup_bulk(qconf->ipv4_lookup_struct, &key_array[0], 4, ret);

    dst_port[0] = (ret[0] < 0)? portid:ipv4_l3fwd_out_if[ret[0]];
    dst_port[1] = (ret[1] < 0)? portid:ipv4_l3fwd_out_if[ret[1]];
    dst_port[2] = (ret[2] < 0)? portid:ipv4_l3fwd_out_if[ret[2]];
    dst_port[3] = (ret[3] < 0)? portid:ipv4_l3fwd_out_if[ret[3]];

    // ...
}
```

The simple_ipv6_fwd_4pkts() function is similar to the simple_ipv4_fwd_4pkts() function.

Known issue: IP packets with extensions or IP packets which are not TCP/UDP cannot work well at this mode.

Packet Forwarding for LPM-based Lookups

For each input packet, the packet forwarding operation is done by the l3fwd_simple_forward() function, but the packet forwarding decision (that is, the identification of the output interface for the packet) for LPM-based lookups is done by the get_ipv4_dst_port() function below:

```
static inline uint16_t
get_ipv4_dst_port(struct rte_ipv4_hdr *ipv4_hdr, uint16_t portid, lookup_struct_t *ipv4_l3fwd_
↪lookup_struct)
{
    uint8_t next_hop;

    return ((rte_lpm_lookup(ipv4_l3fwd_lookup_struct, rte_be_to_cpu_32(ipv4_hdr->dst_addr), &
↪next_hop) == 0)? next_hop : portid);
}
```

Eventdev Driver Initialization

Eventdev driver initialization is same as L2 forwarding eventdev application. Refer *L2 Forwarding Eventdev Sample Application* for more details.

4.22 L3 Forwarding Graph Sample Application

The L3 Forwarding Graph application is a simple example of packet processing using the DPDK Graph framework. The application performs L3 forwarding using Graph framework and nodes written for graph framework.

4.22.1 Overview

The application demonstrates the use of the graph framework and graph nodes `ethdev_rx`, `ip4_lookup`, `ip4_rewrite`, `ethdev_tx` and `pkt_drop` in DPDK to implement packet forwarding.

The initialization is very similar to those of the *L3 Forwarding Sample Application*. There is also additional initialization of graph for graph object creation and configuration per lcore. Run-time path is main thing that differs from L3 forwarding sample application. Difference is that forwarding logic starting from Rx, followed by LPM lookup, TTL update and finally Tx is implemented inside graph nodes. These nodes are interconnected in graph framework. Application main loop needs to walk over graph using `rte_graph_walk()` with graph objects created one per slave lcore.

The lookup method is as per implementation of `ip4_lookup` graph node. The ID of the output interface for the input packet is the next hop returned by the LPM lookup. The set of LPM rules used by the application is statically configured and provided to `ip4_lookup` graph node and `ip4_rewrite` graph node using node control API `rte_node_ip4_route_add()` and `rte_node_ip4_rewrite_add()`.

In the sample application, only IPv4 forwarding is supported as of now.

4.22.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l3fwd-graph` sub-directory.

4.22.3 Running the Application

The application has a number of command line options similar to l3fwd:

```
./l3fwd-graph [EAL options] -- -p PORTMASK
                               [-P]
                               --config(port,queue,lcore)[,(port,queue,lcore)]
                               [--eth-dest=X,MM:MM:MM:MM:MM:MM]
                               [--enable-jumbo [--max-pkt-len PKTLEN]]
                               [--no-numa]
                               [--per-port-pool]
```

Where,

- `-p PORTMASK:` Hexadecimal bitmask of ports to configure

- `-P:` Optional, sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- `--config (port,queue,lcore)[,(port,queue,lcore)]:` Determines which queues from which ports are mapped to which cores.
- `--eth-dest=X,MM:MM:MM:MM:MM:MM:` Optional, ethernet destination for port X.
- `--enable-jumbo:` Optional, enables jumbo frames.
- `--max-pkt-len:` Optional, under the premise of enabling jumbo, maximum packet length in decimal (64-9600).
- `--no-numa:` Optional, disables numa awareness.
- `--per-port-pool:` Optional, set to use independent buffer pools per port. Without this option, single buffer pool is used for all ports.

For example, consider a dual processor socket platform with 8 physical cores, where cores 0-7 and 16-23 appear on socket 0, while cores 8-15 and 24-31 appear on socket 1.

To enable L3 forwarding between two ports, assuming that both ports are in the same socket, using two cores, cores 1 and 2, (which are in the same socket too), use the following command:

```
./build/l3fwd-graph -l 1,2 -n 4 -- -p 0x3 --config="(0,0,1),(1,0,2)"
```

In this command:

- The -l option enables cores 1, 2
- The -p option enables ports 0 and 1
- The –config option enables one queue on each port and maps each (port,queue) pair to a specific core. The following table shows the mapping in this example:

Port	Queue	lcore	Description
0	0	1	Map queue 0 from port 0 to lcore 1.
1	0	2	Map queue 0 from port 1 to lcore 2.

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.22.4 Explanation

The following sections provide some explanation of the sample application code. As mentioned in the overview section, the initialization is similar to that of the *L3 Forwarding Sample Application*. Runtime path though similar in functionality to that of *L3 Forwarding Sample Application*, major part of the implementation is in graph nodes via used via `librte_node` library. The following sections describe aspects that are specific to the L3 Forwarding Graph sample application.

Graph Node Pre-Init Configuration

After device configuration and device Rx, Tx queue setup is complete, a minimal config of port id, num_rx_queues, num_tx_queues, mempools etc will be passed to *ethdev_** node ctrl API `rte_node_eth_config()`. This will be lead to the clone of `ethdev_rx` and `ethdev_tx` nodes as `ethdev_rx-X-Y` and `ethdev_tx-X` where X, Y represent port id and queue id associated with them. In case of `ethdev_tx-X` nodes, tx queue id assigned per instance of the node is same as graph id.

These cloned nodes along with existing static nodes such as `ip4_lookup` and `ip4_rewrite` will be used in graph creation to associate node's to lcore specific graph object.

```
RTE_ETH_FOREACH_DEV(portid)
{

    /* ... */
    ret = rte_eth_dev_configure(portid, nb_rx_queue,
                                n_tx_queue, &local_port_conf);
    /* ... */

    /* Init one TX queue per couple (lcore,port) */
    queueid = 0;
    for (lcore_id = 0; lcore_id < RTE_MAX_LCORE; lcore_id++) {
        /* ... */
        ret = rte_eth_tx_queue_setup(portid, queueid, nb_txd,
                                     socketid, txconf);
        /* ... */
        queueid++;
    }

    /* Setup ethdev node config */
    ethdev_conf[nb_conf].port_id = portid;
    ethdev_conf[nb_conf].num_rx_queues = nb_rx_queue;
    ethdev_conf[nb_conf].num_tx_queues = n_tx_queue;
    if (!per_port_pool)
        ethdev_conf[nb_conf].mp = pktmbuf_pool[0];
    else
      ethdev_conf[nb_conf].mp = pktmbuf_pool[portid];
    ethdev_conf[nb_conf].mp_count = NB_SOCKETS;

    nb_conf++;
    printf("\n");
}

for (lcore_id = 0; lcore_id < RTE_MAX_LCORE; lcore_id++) {
    /* Init RX queues */
    for (queue = 0; queue < qconf->n_rx_queue; ++queue) {
        /* ... */
        if (!per_port_pool)
            ret = rte_eth_rx_queue_setup(portid, queueid, nb_rxd, socketid,
                                         &rxq_conf, pktmbuf_pool[0][socketid]);
        else
          ret = rte_eth_rx_queue_setup(portid, queueid, nb_rxd, socketid,
                                       &rxq_conf, pktmbuf_pool[portid][socketid]);
        /* ... */
    }
}

/* Ethdev node config, skip rx queue mapping */
ret = rte_node_eth_config(ethdev_conf, nb_conf, nb_graphs);
```

Graph Initialization

Now a graph needs to be created with a specific set of nodes for every lcore. A graph object returned after graph creation is a per lcore object and cannot be shared between lcores. Since `ethdev_tx-X` node is per port node, it can be associated with all the graphs created as all the lcores should have Tx capability for every port. But `ethdev_rx-X-Y` node is created per (port, rx_queue_id), so they should be associated with a graph based on the application argument `--config` specifying rx queue mapping to lcore.

Note: The Graph creation will fail if the passed set of shell node pattern's are not sufficient to meet their inter-dependency or even one node is not found with a given regex node pattern.

```
static const char *const default_patterns[] = {
    "ip4*",
    "ethdev_tx-*",
    "pkt_drop",
};
const char **node_patterns;
uint16_t nb_pattern;

/* ... */

/* Create a graph object per lcore with common nodes and
 * lcore specific nodes based on application arguments
 */
nb_patterns = RTE_DIM(default_patterns);
node_patterns = malloc((MAX_RX_QUEUE_PER_LCORE + nb_patterns) *
                       sizeof(*node_patterns));
memcpy(node_patterns, default_patterns,
       nb_patterns * sizeof(*node_patterns));

memset(&graph_conf, 0, sizeof(graph_conf));

/* Common set of nodes in every lcore's graph object */
graph_conf.node_patterns = node_patterns;

for (lcore_id = 0; lcore_id < RTE_MAX_LCORE; lcore_id++) {
    /* ... */

    /* Skip graph creation if no source exists */
    if (!qconf->n_rx_queue)
        continue;

    /* Add rx node patterns of this lcore based on --config */
    for (i = 0; i < qconf->n_rx_queue; i++) {
        graph_conf.node_patterns[nb_patterns + i] =
                            qconf->rx_queue_list[i].node_name;
    }

    graph_conf.nb_node_patterns = nb_patterns + i;
    graph_conf.socket_id = rte_lcore_to_socket_id(lcore_id);

    snprintf(qconf->name, sizeof(qconf->name), "worker_%u", lcore_id);

    graph_id = rte_graph_create(qconf->name, &graph_conf);

    /* ... */

    qconf->graph = rte_graph_lookup(qconf->name);
```

(continues on next page)

(continued from previous page)

```
    /* ... */
}
```

Forwarding data(Route, Next-Hop) addition

Once graph objects are created, node specific info like routes and rewrite headers will be provided run-time using `rte_node_ip4_route_add()` and `rte_node_ip4_rewrite_add()` API.

Note: Since currently `ip4_lookup` and `ip4_rewrite` nodes don't support lock-less mechanisms(RCU, etc) to add run-time forwarding data like route and rewrite data, forwarding data is added before packet processing loop is launched on slave lcore.

```
/* Add route to ip4 graph infra */
for (i = 0; i < IPV4_L3FWD_LPM_NUM_ROUTES; i++) {
    /* ... */

    dst_port = ipv4_l3fwd_lpm_route_array[i].if_out;
    next_hop = i;

    /* ... */
    ret = rte_node_ip4_route_add(ipv4_l3fwd_lpm_route_array[i].ip,
                                 ipv4_l3fwd_lpm_route_array[i].depth, next_hop,
                                 RTE_NODE_IP4_LOOKUP_NEXT_REWRITE);

    /* ... */

    memcpy(rewrite_data, val_eth + dst_port, rewrite_len);

    /* Add next hop for a given destination */
    ret = rte_node_ip4_rewrite_add(next_hop, rewrite_data,
                                   rewrite_len, dst_port);

    RTE_LOG(INFO, L3FWD_GRAPH, "Added route %s, next_hop %u\n",
            route_str, next_hop);
}
```

Packet Forwarding using Graph Walk

Now that all the device configurations are done, graph creations are done and forwarding data is updated with nodes, slave lcores will be launched with graph main loop. Graph main loop is very simple in the sense that it needs to continuously call a non-blocking API `rte_graph_walk()` with it's lcore specific graph object that was already created.

Note: rte_graph_walk() will walk over all the sources nodes i.e `ethdev_rx-X-Y` associated with a given graph and Rx the available packets and enqueue them to the following node `ip4_lookup` which then will enqueue them to `ip4_rewrite` node if LPM lookup succeeds. `ip4_rewrite` node then will update Ethernet header as per next-hop data and transmit the packet via port 'Z' by enqueuing to `ethdev_tx-Z` node instance in its graph object.

```
/* Main processing loop */
static int
graph_main_loop(void *conf)
{
    // ...

    lcore_id = rte_lcore_id();
    qconf = &lcore_conf[lcore_id];
    graph = qconf->graph;

    RTE_LOG(INFO, L3FWD_GRAPH,
            "Entering main loop on lcore %u, graph %s(%p)\n", lcore_id,
            qconf->name, graph);

    /* Walk over graph until signal to quit */
    while (likely(!force_quit))
        rte_graph_walk(graph);
    return 0;
}
```

4.23 L3 Forwarding with Power Management Sample Application

4.23.1 Introduction

The L3 Forwarding with Power Management application is an example of power-aware packet processing using the DPDK. The application is based on existing L3 Forwarding sample application, with the power management algorithms to control the P-states and C-states of the Intel processor via a power management library.

4.23.2 Overview

The application demonstrates the use of the Power libraries in the DPDK to implement packet forwarding. The initialization and run-time paths are very similar to those of the *L3 Forwarding Sample Application*. The main difference from the L3 Forwarding sample application is that this application introduces power-aware optimization algorithms by leveraging the Power library to control P-state and C-state of processor based on packet load.

The DPDK includes poll-mode drivers to configure Intel NIC devices and their receive (Rx) and transmit (Tx) queues. The design principle of this PMD is to access the Rx and Tx descriptors directly without any interrupts to quickly receive, process and deliver packets in the user space.

In general, the DPDK executes an endless packet processing loop on dedicated IA cores that include the following steps:

- Retrieve input packets through the PMD to poll Rx queue
- Process each received packet or provide received packets to other processing cores through software queues
- Send pending output packets to Tx queue through the PMD

In this way, the PMD achieves better performance than a traditional interrupt-mode driver, at the cost of keeping cores active and running at the highest frequency, hence consuming the maximum power all the time. However, during the period of processing light network traffic, which happens regularly in

communication infrastructure systems due to well-known "tidal effect", the PMD is still busy waiting for network packets, which wastes a lot of power.

Processor performance states (P-states) are the capability of an Intel processor to switch between different supported operating frequencies and voltages. If configured correctly, according to system workload, this feature provides power savings. CPUFreq is the infrastructure provided by the Linux* kernel to control the processor performance state capability. CPUFreq supports a user space governor that enables setting frequency via manipulating the virtual file device from a user space application. The Power library in the DPDK provides a set of APIs for manipulating a virtual file device to allow user space application to set the CPUFreq governor and set the frequency of specific cores.

This application includes a P-state power management algorithm to generate a frequency hint to be sent to CPUFreq. The algorithm uses the number of received and available Rx packets on recent polls to make a heuristic decision to scale frequency up/down. Specifically, some thresholds are checked to see whether a specific core running an DPDK polling thread needs to increase frequency a step up based on the near to full trend of polled Rx queues. Also, it decreases frequency a step if packet processed per loop is far less than the expected threshold or the thread's sleeping time exceeds a threshold.

C-States are also known as sleep states. They allow software to put an Intel core into a low power idle state from which it is possible to exit via an event, such as an interrupt. However, there is a tradeoff between the power consumed in the idle state and the time required to wake up from the idle state (exit latency). Therefore, as you go into deeper C-states, the power consumed is lower but the exit latency is increased. Each C-state has a target residency. It is essential that when entering into a C-state, the core remains in this C-state for at least as long as the target residency in order to fully realize the benefits of entering the C-state. CPUIdle is the infrastructure provide by the Linux kernel to control the processor C-state capability. Unlike CPUFreq, CPUIdle does not provide a mechanism that allows the application to change C-state. It actually has its own heuristic algorithms in kernel space to select target C-state to enter by executing privileged instructions like HLT and MWAIT, based on the speculative sleep duration of the core. In this application, we introduce a heuristic algorithm that allows packet processing cores to sleep for a short period if there is no Rx packet received on recent polls. In this way, CPUIdle automatically forces the corresponding cores to enter deeper C-states instead of always running to the C0 state waiting for packets.

Note: To fully demonstrate the power saving capability of using C-states, it is recommended to enable deeper C3 and C6 states in the BIOS during system boot up.

4.23.3 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l3fwd-power` sub-directory.

4.23.4 Running the Application

The application has a number of command line options:

```
./build/l3fwd_power [EAL options] -- -p PORTMASK [-P]  --config(port,queue,lcore)[,(port,queue,
↪lcore)] [--enable-jumbo [--max-pkt-len PKTLEN]] [--no-numa]
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -P: Sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- –config (port,queue,lcore)[,(port,queue,lcore)]: determines which queues from which ports are mapped to which cores.
- –enable-jumbo: optional, enables jumbo frames
- –max-pkt-len: optional, maximum packet length in decimal (64-9600)
- –no-numa: optional, disables numa awareness
- –empty-poll: Traffic Aware power management. See below for details
- –telemetry: Telemetry mode.

See *L3 Forwarding Sample Application* for details. The L3fwd-power example reuses the L3fwd command line options.

4.23.5 Explanation

The following sections provide some explanation of the sample application code. As mentioned in the overview section, the initialization and run-time paths are identical to those of the L3 forwarding application. The following sections describe aspects that are specific to the L3 Forwarding with Power Management sample application.

Power Library Initialization

The Power library is initialized in the main routine. It changes the P-state governor to userspace for specific cores that are under control. The Timer library is also initialized and several timers are created later on, responsible for checking if it needs to scale down frequency at run time by checking CPU utilization statistics.

Note: Only the power management related initialization is shown.

```
int main(int argc, char **argv)
{
    struct lcore_conf *qconf;
    int ret;
    unsigned nb_ports;
    uint16_t queueid, portid;
    unsigned lcore_id;
```

(continues on next page)

(continued from previous page)

```
    uint64_t hz;
    uint32_t n_tx_queue, nb_lcores;
    uint8_t nb_rx_queue, queue, socketid;

    // ...

    /* init RTE timer library to be used to initialize per-core timers */

    rte_timer_subsystem_init();

    // ...


    /* per-core initialization */

    for (lcore_id = 0; lcore_id < RTE_MAX_LCORE; lcore_id++) {
        if (rte_lcore_is_enabled(lcore_id) == 0)
            continue;

        /* init power management library for a specified core */

        ret = rte_power_init(lcore_id);
        if (ret)
            rte_exit(EXIT_FAILURE, "Power management library "
                "initialization failed on core%d\n", lcore_id);

        /* init timer structures for each enabled lcore */

        rte_timer_init(&power_timers[lcore_id]);

        hz = rte_get_hpet_hz();

        rte_timer_reset(&power_timers[lcore_id], hz/TIMER_NUMBER_PER_SECOND, SINGLE, lcore_id,␣
↪power_timer_cb, NULL);

        // ...
    }

    // ...
}
```

Monitoring Loads of Rx Queues

In general, the polling nature of the DPDK prevents the OS power management subsystem from knowing if the network load is actually heavy or light. In this sample, sampling network load work is done by monitoring received and available descriptors on NIC Rx queues in recent polls. Based on the number of returned and available Rx descriptors, this example implements algorithms to generate frequency scaling hints and speculative sleep duration, and use them to control P-state and C-state of processors via the power management library. Frequency (P-state) control and sleep state (C-state) control work individually for each logical core, and the combination of them contributes to a power efficient packet processing solution when serving light network loads.

The rte_eth_rx_burst() function and the newly-added rte_eth_rx_queue_count() function are used in the endless packet processing loop to return the number of received and available Rx descriptors. And those numbers of specific queue are passed to P-state and C-state heuristic algorithms to generate hints based on recent network load trends.

Note: Only power control related code is shown.

```
static
__rte_noreturn int main_loop(__rte_unused void *dummy)
{
    // ...

    while (1) {
    // ...

    /**
     * Read packet from RX queues
     */

    lcore_scaleup_hint = FREQ_CURRENT;
    lcore_rx_idle_count = 0;

    for (i = 0; i < qconf->n_rx_queue; ++i)
    {
        rx_queue = &(qconf->rx_queue_list[i]);
        rx_queue->idle_hint = 0;
        portid = rx_queue->port_id;
        queueid = rx_queue->queue_id;

        nb_rx = rte_eth_rx_burst(portid, queueid, pkts_burst, MAX_PKT_BURST);
        stats[lcore_id].nb_rx_processed += nb_rx;

        if (unlikely(nb_rx == 0)) {
            /**
             * no packet received from rx queue, try to
             * sleep for a while forcing CPU enter deeper
             * C states.
             */

            rx_queue->zero_rx_packet_count++;

            if (rx_queue->zero_rx_packet_count <= MIN_ZERO_POLL_COUNT)
                continue;

            rx_queue->idle_hint = power_idle_heuristic(rx_queue->zero_rx_packet_count);
            lcore_rx_idle_count++;
        } else {
            rx_ring_length = rte_eth_rx_queue_count(portid, queueid);

            rx_queue->zero_rx_packet_count = 0;

            /**
             * do not scale up frequency immediately as
             * user to kernel space communication is costly
             * which might impact packet I/O for received
             * packets.
             */

            rx_queue->freq_up_hint = power_freq_scaleup_heuristic(lcore_id, rx_ring_length);
        }

        /* Prefetch and forward packets */

        // ...
    }
```

(continues on next page)

(continued from previous page)

```
    if (likely(lcore_rx_idle_count != qconf->n_rx_queue)) {
        for (i = 1, lcore_scaleup_hint = qconf->rx_queue_list[0].freq_up_hint; i < qconf->n_rx_
↪queue; ++i) {
            x_queue = &(qconf->rx_queue_list[i]);

            if (rx_queue->freq_up_hint > lcore_scaleup_hint)

                lcore_scaleup_hint = rx_queue->freq_up_hint;
        }

        if (lcore_scaleup_hint == FREQ_HIGHEST)

            rte_power_freq_max(lcore_id);

        else if (lcore_scaleup_hint == FREQ_HIGHER)
            rte_power_freq_up(lcore_id);
        } else {
            /**
             *  All Rx queues empty in recent consecutive polls,
             *  sleep in a conservative manner, meaning sleep as
             * less as possible.
             */

            for (i = 1, lcore_idle_hint = qconf->rx_queue_list[0].idle_hint; i < qconf->n_rx_
↪queue; ++i) {
                rx_queue = &(qconf->rx_queue_list[i]);
                if (rx_queue->idle_hint < lcore_idle_hint)
                    lcore_idle_hint = rx_queue->idle_hint;
            }

            if ( lcore_idle_hint < SLEEP_GEAR1_THRESHOLD)
                /**
                 *   execute "pause" instruction to avoid context
                 *   switch for short sleep.
                 */
                rte_delay_us(lcore_idle_hint);
            else
                /* long sleep force ruining thread to suspend */
                usleep(lcore_idle_hint);

           stats[lcore_id].sleep_time += lcore_idle_hint;
        }
    }
}
```

P-State Heuristic Algorithm

The power_freq_scaleup_heuristic() function is responsible for generating a frequency hint for the specified logical core according to available descriptor number returned from rte_eth_rx_queue_count(). On every poll for new packets, the length of available descriptor on an Rx queue is evaluated, and the algorithm used for frequency hinting is as follows:

- If the size of available descriptors exceeds 96, the maximum frequency is hinted.
- If the size of available descriptors exceeds 64, a trend counter is incremented by 100.
- If the length of the ring exceeds 32, the trend counter is incremented by 1.
- When the trend counter reached 10000 the frequency hint is changed to the next higher frequency.

Note: The assumption is that the Rx queue size is 128 and the thresholds specified above must be adjusted accordingly based on actual hardware Rx queue size, which are configured via the rte_eth_rx_queue_setup() function.

In general, a thread needs to poll packets from multiple Rx queues. Most likely, different queue have different load, so they would return different frequency hints. The algorithm evaluates all the hints and then scales up frequency in an aggressive manner by scaling up to highest frequency as long as one Rx queue requires. In this way, we can minimize any negative performance impact.

On the other hand, frequency scaling down is controlled in the timer callback function. Specifically, if the sleep times of a logical core indicate that it is sleeping more than 25% of the sampling period, or if the average packet per iteration is less than expectation, the frequency is decreased by one step.

C-State Heuristic Algorithm

Whenever recent rte_eth_rx_burst() polls return 5 consecutive zero packets, an idle counter begins incrementing for each successive zero poll. At the same time, the function power_idle_heuristic() is called to generate speculative sleep duration in order to force logical to enter deeper sleeping C-state. There is no way to control C- state directly, and the CPUIdle subsystem in OS is intelligent enough to select C-state to enter based on actual sleep period time of giving logical core. The algorithm has the following sleeping behavior depending on the idle counter:

- If idle count less than 100, the counter value is used as a microsecond sleep value through rte_delay_us() which execute pause instructions to avoid costly context switch but saving power at the same time.
- If idle count is between 100 and 999, a fixed sleep interval of 100 µs is used. A 100 µs sleep interval allows the core to enter the C1 state while keeping a fast response time in case new traffic arrives.
- If idle count is greater than 1000, a fixed sleep value of 1 ms is used until the next timer expiration is used. This allows the core to enter the C3/C6 states.

Note: The thresholds specified above need to be adjusted for different Intel processors and traffic profiles.

If a thread polls multiple Rx queues and different queue returns different sleep duration values, the algorithm controls the sleep time in a conservative manner by sleeping for the least possible time in order to avoid a potential performance impact.

4.23.6 Empty Poll Mode

Additionally, there is a traffic aware mode of operation called "Empty Poll" where the number of empty polls can be monitored to keep track of how busy the application is. Empty poll mode can be enabled by the command line option –empty-poll.

See *Power Management* chapter in the DPDK Programmer's Guide for empty poll mode details.

```
./l3fwd-power -l xxx   -n 4   -w 0000:xx:00.0 -w 0000:xx:00.1 -- -p 0x3 -P --config="(0,0,xx),
↪(1,0,xx)" --empty-poll="0,0,0" -l 14 -m 9 -h 1
```

Where,

–empty-poll: Enable the empty poll mode instead of original algorithm

–empty-poll="training_flag, med_threshold, high_threshold"

- `training_flag` : optional, enable/disable training mode. Default value is 0. If the training_flag is set as 1(true), then the application will start in training mode and print out the trained threshold values. If the training_flag is set as 0(false), the application will start in normal mode, and will use either the default thresholds or those supplied on the command line. The trained threshold values are specific to the user's system, may give a better power profile when compared to the default threshold values.
- `med_threshold` : optional, sets the empty poll threshold of a modestly busy system state. If this is not supplied, the application will apply the default value of 350000.
- `high_threshold` : optional, sets the empty poll threshold of a busy system state. If this is not supplied, the application will apply the default value of 580000.
- -l : optional, set up the LOW power state frequency index
- -m : optional, set up the MED power state frequency index
- -h : optional, set up the HIGH power state frequency index

Empty Poll Mode Example Usage

To initially obtain the ideal thresholds for the system, the training mode should be run first. This is achieved by running the l3fwd-power app with the training flag set to "1", and the other parameters set to 0.

```
./examples/l3fwd-power/build/l3fwd-power -l 1-3 -- -p 0x0f --config="(0,0,2),(0,1,3)" --empty-
↪poll "1,0,0" -P
```

This will run the training algorithm for x seconds on each core (cores 2 and 3), and then print out the recommended threshold values for those cores. The thresholds should be very similar for each core.

```
POWER: Bring up the Timer
POWER: set the power freq to MED
POWER: Low threshold is 230277
POWER: MED threshold is 335071
POWER: HIGH threshold is 523769
POWER: Training is Complete for 2
POWER: set the power freq to MED
POWER: Low threshold is 236814
POWER: MED threshold is 344567
POWER: HIGH threshold is 538580
POWER: Training is Complete for 3
```

Once the values have been measured for a particular system, the app can then be started without the training mode so traffic can start immediately.

```
./examples/l3fwd-power/build/l3fwd-power -l 1-3 -- -p 0x0f --config="(0,0,2),(0,1,3)" --empty-
↪poll "0,340000,540000" -P
```

4.23.7 Telemetry Mode

The telemetry mode support for `l3fwd-power` is a standalone mode, in this mode `l3fwd-power` does simple l3fwding along with calculating empty polls, full polls, and busy percentage for each forwarding core. The aggregation of these values of all cores is reported as application level telemetry to metric library for every 500ms from the master core.

The busy percentage is calculated by recording the poll_count and when the count reaches a defined value the total cycles it took is measured and compared with minimum and maximum reference cycles and accordingly busy rate is set to either 0% or 50% or 100%.

Note:

- The CONFIG_RTE_LIBRTE_TELEMETRY should be set in order to get the stats in DPDK telemetry.

```
./examples/l3fwd-power/build/l3fwd-power --telemetry -l 1-3 -- -p 0x0f --config="(0,0,2),(0,1,
↪3)" --telemetry
```

The new stats `empty_poll` , `full_poll` and `busy_percent` can be viewed by running the script `/usertools/dpdk-telemetry-client.py` and selecting the menu option `Send for global Metrics`.

4.24 L3 Forwarding with Access Control Sample Application

The L3 Forwarding with Access Control application is a simple example of packet processing using the DPDK. The application performs a security check on received packets. Packets that are in the Access Control List (ACL), which is loaded during initialization, are dropped. Others are forwarded to the correct port.

4.24.1 Overview

The application demonstrates the use of the ACL library in the DPDK to implement access control and packet L3 forwarding. The application loads two types of rules at initialization:

- Route information rules, which are used for L3 forwarding
- Access Control List (ACL) rules that blacklist (or block) packets with a specific characteristic

When packets are received from a port, the application extracts the necessary information from the TCP/IP header of the received packet and performs a lookup in the rule database to figure out whether the packets should be dropped (in the ACL range) or forwarded to desired ports. The initialization and run-time paths are similar to those of the *L3 Forwarding Sample Application*. However, there are significant differences in the two applications. For example, the original L3 forwarding application uses either LPM or an exact match algorithm to perform forwarding port lookup, while this application uses the ACL library to perform both ACL and route entry lookup. The following sections provide more detail.

Classification for both IPv4 and IPv6 packets is supported in this application. The application also assumes that all the packets it processes are TCP/UDP packets and always extracts source/destination port information from the packets.

Tuple Packet Syntax

The application implements packet classification for the IPv4/IPv6 5-tuple syntax specifically. The 5-tuple syntax consist of a source IP address, a destination IP address, a source port, a destination port and a protocol identifier. The fields in the 5-tuple syntax have the following formats:

- **Source IP address and destination IP address** : Each is either a 32-bit field (for IPv4), or a set of 4 32-bit fields (for IPv6) represented by a value and a mask length. For example, an IPv4 range of 192.168.1.0 to 192.168.1.255 could be represented by a value = [192, 168, 1, 0] and a mask length = 24.
- **Source port and destination port** : Each is a 16-bit field, represented by a lower start and a higher end. For example, a range of ports 0 to 8192 could be represented by lower = 0 and higher = 8192.
- **Protocol identifier** : An 8-bit field, represented by a value and a mask, that covers a range of values. To verify that a value is in the range, use the following expression: "(VAL & mask) == value"

The trick in how to represent a range with a mask and value is as follows. A range can be enumerated in binary numbers with some bits that are never changed and some bits that are dynamically changed. Set those bits that dynamically changed in mask and value with 0. Set those bits that never changed in the mask with 1, in value with number expected. For example, a range of 6 to 7 is enumerated as 0b110 and 0b111. Bit 1-7 are bits never changed and bit 0 is the bit dynamically changed. Therefore, set bit 0 in mask and value with 0, set bits 1-7 in mask with 1, and bits 1-7 in value with number 0b11. So, mask is 0xfe, value is 0x6.

Note: The library assumes that each field in the rule is in LSB or Little Endian order when creating the database. It internally converts them to MSB or Big Endian order. When performing a lookup, the library assumes the input is in MSB or Big Endian order.

Access Rule Syntax

In this sample application, each rule is a combination of the following:

- 5-tuple field: This field has a format described in Section.
- priority field: A weight to measure the priority of the rules. The rule with the higher priority will ALWAYS be returned if the specific input has multiple matches in the rule database. Rules with lower priority will NEVER be returned in any cases.
- userdata field: A user-defined field that could be any value. It can be the forwarding port number if the rule is a route table entry or it can be a pointer to a mapping address if the rule is used for address mapping in the NAT application. The key point is that it is a useful reserved field for user convenience.

ACL and Route Rules

The application needs to acquire ACL and route rules before it runs. Route rules are mandatory, while ACL rules are optional. To simplify the complexity of the priority field for each rule, all ACL and route entries are assumed to be in the same file. To read data from the specified file successfully, the application assumes the following:

- Each rule occupies a single line.
- Only the following four rule line types are valid in this application:
- ACL rule line, which starts with a leading character '@'
- Route rule line, which starts with a leading character 'R'
- Comment line, which starts with a leading character '#'
- Empty line, which consists of a space, form-feed ('f'), newline ('n'), carriage return ('r'), horizontal tab ('t'), or vertical tab ('v').

Other lines types are considered invalid.

- Rules are organized in descending order of priority, which means rules at the head of the file always have a higher priority than those further down in the file.
- A typical IPv4 ACL rule line should have a format as shown below:

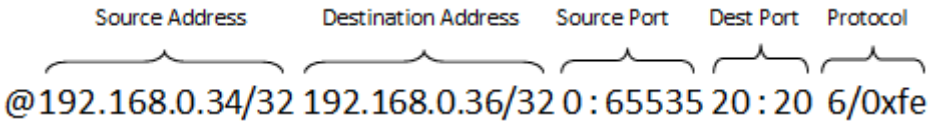


Fig. 4.9: A typical IPv4 ACL rule

IPv4 addresses are specified in CIDR format as specified in RFC 4632. They consist of the dot notation for the address and a prefix length separated by '/'. For example, 192.168.0.34/32, where the address is 192.168.0.34 and the prefix length is 32.

Ports are specified as a range of 16-bit numbers in the format MIN:MAX, where MIN and MAX are the inclusive minimum and maximum values of the range. The range 0:65535 represents all possible ports in a range. When MIN and MAX are the same value, a single port is represented, for example, 20:20.

The protocol identifier is an 8-bit value and a mask separated by '/'. For example: 6/0xfe matches protocol values 6 and 7.

- Route rules start with a leading character 'R' and have the same format as ACL rules except an extra field at the tail that indicates the forwarding port number.

Rules File Example

Each rule is explained as follows:

- Rule 1 (the first line) tells the application to drop those packets with source IP address = [1.2.3.*], destination IP address = [192.168.0.36], protocol = [6]/[7]
- Rule 2 (the second line) is similar to Rule 1, except the source IP address is ignored. It tells the application to forward packets with destination IP address = [192.168.0.36], protocol = [6]/[7], destined to port 1.

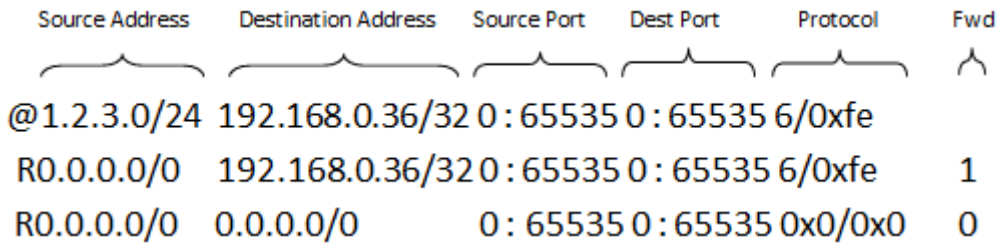


Fig. 4.10: Rules example

- Rule 3 (the third line) tells the application to forward all packets to port 0. This is something like a default route entry.

As described earlier, the application assume rules are listed in descending order of priority, therefore Rule 1 has the highest priority, then Rule 2, and finally, Rule 3 has the lowest priority.

Consider the arrival of the following three packets:

- Packet 1 has source IP address = [1.2.3.4], destination IP address = [192.168.0.36], and protocol = [6]
- Packet 2 has source IP address = [1.2.4.4], destination IP address = [192.168.0.36], and protocol = [6]
- Packet 3 has source IP address = [1.2.3.4], destination IP address = [192.168.0.36], and protocol = [8]

Observe that:

- Packet 1 matches all of the rules
- Packet 2 matches Rule 2 and Rule 3
- Packet 3 only matches Rule 3

For priority reasons, Packet 1 matches Rule 1 and is dropped. Packet 2 matches Rule 2 and is forwarded to port 1. Packet 3 matches Rule 3 and is forwarded to port 0.

For more details on the rule file format, please refer to rule_ipv4.db and rule_ipv6.db files (inside <RTE_SDK>/examples/l3fwd-acl/).

Application Phases

Once the application starts, it transitions through three phases:

- **Initialization Phase** - Perform the following tasks:
- Parse command parameters. Check the validity of rule file(s) name(s), number of logical cores, receive and transmit queues. Bind ports, queues and logical cores. Check ACL search options, and so on.
- Call Environmental Abstraction Layer (EAL) and Poll Mode Driver (PMD) functions to initialize the environment and detect possible NICs. The EAL creates several threads and sets affinity to a specific hardware thread CPU based on the configuration specified by the command line arguments.
- Read the rule files and format the rules into the representation that the ACL library can recognize. Call the ACL library function to add the rules into the database and compile them as a trie of pattern sets. Note that application maintains a separate AC contexts for IPv4 and IPv6 rules.

- **Runtime Phase** - Process the incoming packets from a port. Packets are processed in three steps:
  - Retrieval: Gets a packet from the receive queue. Each logical core may process several queues for different ports. This depends on the configuration specified by command line arguments.
  - Lookup: Checks that the packet type is supported (IPv4/IPv6) and performs a 5-tuple lookup over corresponding AC context. If an ACL rule is matched, the packets will be dropped and return back to step 1. If a route rule is matched, it indicates the packet is not in the ACL list and should be forwarded. If there is no matches for the packet, then the packet is dropped.
  - Forwarding: Forwards the packet to the corresponding port.
- **Final Phase** - Perform the following tasks:

  Calls the EAL, PMD driver and ACL library to free resource, then quits.

4.24.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `l3fwd-acl` sub-directory.

4.24.3 Running the Application

The application has a number of command line options:

```
./build/l3fwd-acl [EAL options] -- -p PORTMASK [-P] --config(port,queue,lcore)[,(port,queue,
↪lcore)] --rule_ipv4 FILENAME rule_ipv6 FILENAME [--scalar] [--enable-jumbo [--max-pkt-len␣
↪PKTLEN]] [--no-numa]
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -P: Sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- –config (port,queue,lcore)[,(port,queue,lcore)]: determines which queues from which ports are mapped to which cores
- –rule_ipv4 FILENAME: Specifies the IPv4 ACL and route rules file
- –rule_ipv6 FILENAME: Specifies the IPv6 ACL and route rules file
- –scalar: Use a scalar function to perform rule lookup
- –enable-jumbo: optional, enables jumbo frames
- –max-pkt-len: optional, maximum packet length in decimal (64-9600)
- –no-numa: optional, disables numa awareness

For example, consider a dual processor socket platform with 8 physical cores, where cores 0-7 and 16-23 appear on socket 0, while cores 8-15 and 24-31 appear on socket 1.

To enable L3 forwarding between two ports, assuming that both ports are in the same socket, using two cores, cores 1 and 2, (which are in the same socket too), use the following command:

```
./build/l3fwd-acl -l 1,2 -n 4 -- -p 0x3 --config="(0,0,1),(1,0,2)" --rule_ipv4="./rule_ipv4.db
↪" -- rule_ipv6="./rule_ipv6.db" --scalar
```

In this command:

- The -l option enables cores 1, 2
- The -p option enables ports 0 and 1
- The –config option enables one queue on each port and maps each (port,queue) pair to a specific core. The following table shows the mapping in this example:

Port	Queue	lcore	Description
0	0	1	Map queue 0 from port 0 to lcore 1.
1	0	2	Map queue 0 from port 1 to lcore 2.

- The –rule_ipv4 option specifies the reading of IPv4 rules sets from the ./ rule_ipv4.db file.
- The –rule_ipv6 option specifies the reading of IPv6 rules sets from the ./ rule_ipv6.db file.
- The –scalar option specifies the performing of rule lookup with a scalar function.

4.24.4 Explanation

The following sections provide some explanation of the sample application code. The aspects of port, device and CPU configuration are similar to those of the *L3 Forwarding Sample Application*. The following sections describe aspects that are specific to L3 forwarding with access control.

Parse Rules from File

As described earlier, both ACL and route rules are assumed to be saved in the same file. The application parses the rules from the file and adds them to the database by calling the ACL library function. It ignores empty and comment lines, and parses and validates the rules it reads. If errors are detected, the application exits with messages to identify the errors encountered.

The application needs to consider the userdata and priority fields. The ACL rules save the index to the specific rules in the userdata field, while route rules save the forwarding port number. In order to differentiate the two types of rules, ACL rules add a signature in the userdata field. As for the priority field, the application assumes rules are organized in descending order of priority. Therefore, the code only decreases the priority number with each rule it parses.

Setting Up the ACL Context

For each supported AC rule format (IPv4 5-tuple, IPv6 6-tuple) application creates a separate context handler from the ACL library for each CPU socket on the board and adds parsed rules into that context.

Note, that for each supported rule type, application needs to calculate the expected offset of the fields from the start of the packet. That's why only packets with fixed IPv4/ IPv6 header are supported. That allows to perform ACL classify straight over incoming packet buffer - no extra protocol field retrieval need to be performed.

Subsequently, the application checks whether NUMA is enabled. If it is, the application records the socket IDs of the CPU cores involved in the task.

Finally, the application creates contexts handler from the ACL library, adds rules parsed from the file into the database and build an ACL trie. It is important to note that the application creates an independent copy of each database for each socket CPU involved in the task to reduce the time for remote memory access.

4.25 Link Status Interrupt Sample Application

The Link Status Interrupt sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) that demonstrates how network link status changes for a network port can be captured and used by a DPDK application.

4.25.1 Overview

The Link Status Interrupt sample application registers a user space callback for the link status interrupt of each port and performs L2 forwarding for each packet that is received on an RX_PORT. The following operations are performed:

- RX_PORT and TX_PORT are paired with available ports one-by-one according to the core mask
- The source MAC address is replaced by the TX_PORT MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

This application can be used to demonstrate the usage of link status interrupt and its user space callbacks and the behavior of L2 forwarding each time the link status changes.

4.25.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `link_status_interrupt` sub-directory.

4.25.3 Running the Application

The application requires a number of command line options:

```
./build/link_status_interrupt [EAL options] -- -p PORTMASK [-q NQ][-T PERIOD]
```

where,

- -p PORTMASK: A hexadecimal bitmask of the ports to configure
- -q NQ: A number of queues (=ports) per lcore (default is 1)
- -T PERIOD: statistics will be refreshed each PERIOD seconds (0 to disable, 10 default)

To run the application in a linux environment with 4 lcores, 4 memory channels, 16 ports and 8 RX queues per lcore, issue the command:

```
$ ./build/link_status_interrupt -l 0-3 -n 4-- -q 8 -p ffff
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.25.4 Explanation

The following sections provide some explanation of the code.

Command Line Arguments

The Link Status Interrupt sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments (see Section *Running the Application*).

Command line parsing is done in the same way as it is done in the L2 Forwarding Sample Application. See *Command Line Arguments* for more information.

Mbuf Pool Initialization

Mbuf pool initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See *Mbuf Pool Initialization* for more information.

Driver Initialization

The main part of the code in the main() function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the Poll Mode Driver in the *DPDK Programmer's Guide and the DPDK API Reference*.

```
/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */

RTE_ETH_FOREACH_DEV(portid) {
    /* skip ports that are not enabled */

    if ((lsi_enabled_port_mask & (1 << portid)) == 0)
        continue;

    /* save the destination port id */

    if (nb_ports_in_mask % 2) {
        lsi_dst_ports[portid] = portid_last;
        lsi_dst_ports[portid_last] = portid;
    }
    else
        portid_last = portid;

    nb_ports_in_mask++;

    rte_eth_dev_info_get((uint8_t) portid, &dev_info);
}
```

The next step is to configure the RX and TX queues. For each port, there is only one RX queue (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. The rte_eth_dev_configure() function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t) portid, 1, 1, &port_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot configure device: err=%d, port=%u\n", ret, portid);
```

The global configuration is stored in a static structure:

```
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .split_hdr_size = 0,
    },
    .txmode = {},
    .intr_conf = {
        .lsc = 1, /**< link status interrupt feature enabled */
    },
};
```

Configuring lsc to 0 (the default) disables the generation of any link status change interrupts in kernel space and no user space interrupt event is received. The public interface rte_eth_link_get() accesses the NIC registers directly to update the link status. Configuring lsc to non-zero enables the generation of link status change interrupts in kernel space when a link status change is present and calls the user space callbacks registered by the application. The public interface rte_eth_link_get() just reads the link status in a global structure that would be updated in the interrupt host thread only.

Interrupt Callback Registration

The application can register one or more callbacks to a specific port and interrupt event. An example callback function that has been written as indicated below.

```
static void
lsi_event_callback(uint16_t port_id, enum rte_eth_event_type type, void *param)
{
    struct rte_eth_link link;
    int ret;

    RTE_SET_USED(param);

    printf("\n\nIn registered callback...\n");

    printf("Event type: %s\n", type == RTE_ETH_EVENT_INTR_LSC ? "LSC interrupt" : "unknown␣
↪event");

    ret = rte_eth_link_get_nowait(port_id, &link);
    if (ret < 0) {
        printf("Failed to get port %d link status: %s\n\n",
               port_id, rte_strerror(-ret));
    } else if (link.link_status) {
        printf("Port %d Link Up - speed %u Mbps - %s\n\n", port_id, (unsigned)link.link_speed,
              (link.link_duplex == ETH_LINK_FULL_DUPLEX) ? ("full-duplex") : ("half-duplex"));
    } else
        printf("Port %d Link Down\n\n", port_id);
}
```

This function is called when a link status interrupt is present for the right port. The port_id indicates which port the interrupt applies to. The type parameter identifies the interrupt event type, which currently can be RTE_ETH_EVENT_INTR_LSC only, but other types can be added in the future. The param parameter is the address of the parameter for the callback. This function should be implemented with care since it will be called in the interrupt host thread, which is different from the main thread of its caller.

The application registers the lsi_event_callback and a NULL parameter to the link status interrupt event on each port:

```
rte_eth_dev_callback_register((uint8_t)portid, RTE_ETH_EVENT_INTR_LSC, lsi_event_callback,␣
↪NULL);
```

This registration can be done only after calling the rte_eth_dev_configure() function and before calling any other function. If lsc is initialized with 0, the callback is never called since no interrupt event would ever be present.

RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the -q option, which specifies the number of queues per lcore.

For example, if the user specifies -q 4, the application is able to poll four ports with one lcore. If there are 16 ports on the target (and if the portmask argument is -p ffff), the application will need four lcores to poll all the ports.

```
ret = rte_eth_rx_queue_setup((uint8_t) portid, 0, nb_rxd, SOCKET0, &rx_conf, lsi_pktmbuf_pool);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_rx_queue_setup: err=%d, port=%u\n", ret, portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called struct lcore_queue_conf.

```
struct lcore_queue_conf {
    unsigned n_rx_port;
    unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE]; unsigned tx_queue_id;
    struct mbuf_table tx_mbufs[LSI_MAX_PORTS];
} rte_cache_aligned;

struct lcore_queue_conf lcore_queue_conf[RTE_MAX_LCORE];
```

The n_rx_port and rx_port_list[] fields are used in the main packet processing loop (see *Receive, Process and Transmit Packets*).

The global configuration for the RX queues is stored in a static structure:

```
static const struct rte_eth_rxconf rx_conf = {
    .rx_thresh = {
        .pthresh = RX_PTHRESH,
        .hthresh = RX_HTHRESH,
        .wthresh = RX_WTHRESH,
    },
};
```

TX Queue Initialization

Each lcore should be able to transmit on any port. For every port, a single TX queue is initialized.

```
/* init one TX queue logical core on each port */

fflush(stdout);

ret = rte_eth_tx_queue_setup(portid, 0, nb_txd, rte_eth_dev_socket_id(portid), &tx_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup: err=%d,port=%u\n", ret, (unsigned) portid);
```

The global configuration for TX queues is stored in a static structure:

```
static const struct rte_eth_txconf tx_conf = {
    .tx_thresh = {
        .pthresh = TX_PTHRESH,
        .hthresh = TX_HTHRESH,
        .wthresh = TX_WTHRESH,
    },
    .tx_free_thresh = RTE_TEST_TX_DESC_DEFAULT + 1, /* disable feature */
};
```

Receive, Process and Transmit Packets

In the lsi_main_loop() function, the main task is to read ingress packets from the RX queues. This is done using the following code:

```
/*
 *   Read packet from RX queues
 */

for (i = 0; i < qconf->n_rx_port; i++) {
    portid = qconf->rx_port_list[i];
    nb_rx = rte_eth_rx_burst((uint8_t) portid, 0, pkts_burst, MAX_PKT_BURST);
    port_statistics[portid].rx += nb_rx;

    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
        rte_prefetch0(rte_pktmbuf_mtod(m, void *));
        lsi_simple_forward(m, portid);
    }
}
```

Packets are read in a burst of size MAX_PKT_BURST. The rte_eth_rx_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

Then, each mbuf in the table is processed by the lsi_simple_forward() function. The processing is very simple: processes the TX port from the RX port and then replaces the source and destination MAC addresses.

Note: In the following code, the two lines for calculating the output port require some explanation. If portId is even, the first line does nothing (as portid & 1 will be 0), and the second line adds 1. If portId is odd, the first line subtracts one and the second line does nothing. Therefore, 0 goes to 1, and 1 to 0, 2 goes to 3 and 3 to 2, and so on.

```
static void
lsi_simple_forward(struct rte_mbuf *m, unsigned portid)
{
    struct rte_ether_hdr *eth;
    void *tmp;
    unsigned dst_port = lsi_dst_ports[portid];

    eth = rte_pktmbuf_mtod(m, struct rte_ether_hdr *);

    /* 02:00:00:00:00:xx */

    tmp = &eth->d_addr.addr_bytes[0];
```

(continues on next page)

(continued from previous page)

```
    *((uint64_t *)tmp) = 0x000000000002 + (dst_port << 40);

    /* src addr */
    rte_ether_addr_copy(&lsi_ports_eth_addr[dst_port], &eth->s_addr);

    lsi_send_packet(m, dst_port);
}
```

Then, the packet is sent using the lsi_send_packet(m, dst_port) function. For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the lsi_send_burst() function directly from the main loop to send all the received packets on the same TX port using the burst-oriented send function, which is more efficient.

However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that so the same approach can be reused in a more complex application.

The lsi_send_packet() function stores the packet in a per-lcore and per-txport table. If the table is full, the whole packets table is transmitted using the lsi_send_burst() function:

```
/* Send the packet on an output interface */

static int
lsi_send_packet(struct rte_mbuf *m, uint16_t port)
{
    unsigned lcore_id, len;
    struct lcore_queue_conf *qconf;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = m;
    len++;

    /* enough pkts to be sent */

    if (unlikely(len == MAX_PKT_BURST)) {
        lsi_send_burst(qconf, MAX_PKT_BURST, port);
        len = 0;
    }
    qconf->tx_mbufs[port].len = len;

    return 0;
}
```

To ensure that no packets remain in the tables, each lcore does a draining of the TX queue in its main loop. This technique introduces some latency when there are not many packets to send. However, it improves performance:

```
 cur_tsc = rte_rdtsc();

 /*
  *    TX burst queue drain
  */

 diff_tsc = cur_tsc - prev_tsc;

 if (unlikely(diff_tsc > drain_tsc)) {
```

(continues on next page)

(continued from previous page)

```
    /* this could be optimized (use queueid instead of * portid), but it is not called so␣
↪often */

    for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
        if (qconf->tx_mbufs[portid].len == 0)
            continue;

        lsi_send_burst(&lcore_queue_conf[lcore_id],
        qconf->tx_mbufs[portid].len, (uint8_t) portid);
        qconf->tx_mbufs[portid].len = 0;
    }

    /* if timer is enabled */

    if (timer_period > 0) {
        /* advance the timer */

        timer_tsc += diff_tsc;

        /* if timer has reached its timeout */

        if (unlikely(timer_tsc >= (uint64_t) timer_period)) {
            /* do this only on master core */

            if (lcore_id == rte_get_master_lcore()) {
                print_stats();

                /* reset the timer */
                timer_tsc = 0;
            }
        }
    }
    prev_tsc = cur_tsc;
}
```

4.26 Server-Node EFD Sample Application

This sample application demonstrates the use of EFD library as a flow-level load balancer, for more information about the EFD Library please refer to the DPDK programmer's guide.

This sample application is a variant of the *client-server sample application* where a specific target node is specified for every and each flow (not in a round-robin fashion as the original load balancing sample application).

4.26.1 Overview

The architecture of the EFD flow-based load balancer sample application is presented in the following figure.

Fig. 4.11: Using EFD as a Flow-Level Load Balancer

As shown in Fig. 4.11, the sample application consists of a front-end node (server) using the EFD library to create a load-balancing table for flows, for each flow a target backend worker node is specified. The

EFD table does not store the flow key (unlike a regular hash table), and hence, it can individually load-balance millions of flows (number of targets * maximum number of flows fit in a flow table per target) while still fitting in CPU cache.

It should be noted that although they are referred to as nodes, the frontend server and worker nodes are processes running on the same platform.

Front-end Server

Upon initializing, the frontend server node (process) creates a flow distributor table (based on the EFD library) which is populated with flow information and its intended target node.

The sample application assigns a specific target node_id (process) for each of the IP destination addresses as follows:

```
node_id = i % num_nodes; /* Target node id is generated */
ip_dst = rte_cpu_to_be_32(i); /* Specific ip destination address is
                                 assigned to this target node */
```

then the pair of <key,target> is inserted into the flow distribution table.

The main loop of the server process receives a burst of packets, then for each packet, a flow key (IP destination address) is extracted. The flow distributor table is looked up and the target node id is returned. Packets are then enqueued to the specified target node id.

It should be noted that flow distributor table is not a membership test table. I.e. if the key has already been inserted the target node id will be correct, but for new keys the flow distributor table will return a value (which can be valid).

Backend Worker Nodes

Upon initializing, the worker node (process) creates a flow table (a regular hash table that stores the key default size 1M flows) which is populated with only the flow information that is serviced at this node. This flow key is essential to point out new keys that have not been inserted before.

The worker node's main loop is simply receiving packets then doing a hash table lookup. If a match occurs then statistics are updated for flows serviced by this node. If no match is found in the local hash table then this indicates that this is a new flow, which is dropped.

4.26.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `server_node_efd` sub-directory.

4.26.3 Running the Application

The application has two binaries to be run: the front-end server and the back-end node.

The frontend server (server) has the following command line options:

```
./server [EAL options] -- -p PORTMASK -n NUM_NODES -f NUM_FLOWS
```

Where,

- `-p PORTMASK:` Hexadecimal bitmask of ports to configure
- `-n NUM_NODES:` Number of back-end nodes that will be used
- `-f NUM_FLOWS:` Number of flows to be added in the EFD table (1 million, by default)

The back-end node (node) has the following command line options:

```
./node [EAL options] -- -n NODE_ID
```

Where,

- `-n NODE_ID:` Node ID, which cannot be equal or higher than NUM_MODES

First, the server app must be launched, with the number of nodes that will be run. Once it has been started, the node instances can be run, with different NODE_ID. These instances have to be run as secondary processes, with `--proc-type=secondary` in the EAL options, which will attach to the primary process memory, and therefore, they can access the queues created by the primary process to distribute packets.

To successfully run the application, the command line used to start the application has to be in sync with the traffic flows configured on the traffic generator side.

For examples of application command lines and traffic generator flows, please refer to the DPDK Test Report. For more details on how to set up and run the sample applications provided with DPDK package, please refer to the *DPDK Getting Started Guide for Linux* and *DPDK Getting Started Guide for FreeBSD*.

4.26.4 Explanation

As described in previous sections, there are two processes in this example.

The first process, the front-end server, creates and populates the EFD table, which is used to distribute packets to nodes, which the number of flows specified in the command line (1 million, by default).

```
static void
create_efd_table(void)
{
    uint8_t socket_id = rte_socket_id();

    /* create table */
    efd_table = rte_efd_create("flow table", num_flows * 2, sizeof(uint32_t),
                    1 << socket_id, socket_id);

    if (efd_table == NULL)
        rte_exit(EXIT_FAILURE, "Problem creating the flow table\n");
}

static void
populate_efd_table(void)
{
```

(continues on next page)

(continued from previous page)

```
    unsigned int i;
    int32_t ret;
    uint32_t ip_dst;
    uint8_t socket_id = rte_socket_id();
    uint64_t node_id;

    /* Add flows in table */
    for (i = 0; i < num_flows; i++) {
        node_id = i % num_nodes;

        ip_dst = rte_cpu_to_be_32(i);
        ret = rte_efd_update(efd_table, socket_id,
                        (void *)&ip_dst, (efd_value_t)node_id);
        if (ret < 0)
            rte_exit(EXIT_FAILURE, "Unable to add entry %u in "
                            "EFD table\n", i);
    }

    printf("EFD table: Adding 0x%x keys\n", num_flows);
}
```

After initialization, packets are received from the enabled ports, and the IPv4 address from the packets is used as a key to look up in the EFD table, which tells the node where the packet has to be distributed.

```
static void
process_packets(uint32_t port_num __rte_unused, struct rte_mbuf *pkts[],
        uint16_t rx_count, unsigned int socket_id)
{
    uint16_t i;
    uint8_t node;
    efd_value_t data[EFD_BURST_MAX];
    const void *key_ptrs[EFD_BURST_MAX];

    struct rte_ipv4_hdr *ipv4_hdr;
    uint32_t ipv4_dst_ip[EFD_BURST_MAX];

    for (i = 0; i < rx_count; i++) {
        /* Handle IPv4 header.*/
        ipv4_hdr = rte_pktmbuf_mtod_offset(pkts[i], struct rte_ipv4_hdr *,
                sizeof(struct rte_ether_hdr));
        ipv4_dst_ip[i] = ipv4_hdr->dst_addr;
        key_ptrs[i] = (void *)&ipv4_dst_ip[i];
    }

    rte_efd_lookup_bulk(efd_table, socket_id, rx_count,
                (const void **) key_ptrs, data);
    for (i = 0; i < rx_count; i++) {
        node = (uint8_t) ((uintptr_t)data[i]);

        if (node >= num_nodes) {
            /*
             * Node is out of range, which means that
             * flow has not been inserted
             */
            flow_dist_stats.drop++;
            rte_pktmbuf_free(pkts[i]);
        } else {
            flow_dist_stats.distributed++;
            enqueue_rx_packet(node, pkts[i]);
        }
```

(continues on next page)

(continued from previous page)

```
    }

    for (i = 0; i < num_nodes; i++)
        flush_rx_queue(i);
}
```

The burst of packets received is enqueued in temporary buffers (per node), and enqueued in the shared ring between the server and the node. After this, a new burst of packets is received and this process is repeated infinitely.

```
static void
flush_rx_queue(uint16_t node)
{
    uint16_t j;
    struct node *cl;

    if (cl_rx_buf[node].count == 0)
        return;

    cl = &nodes[node];
    if (rte_ring_enqueue_bulk(cl->rx_q, (void **)cl_rx_buf[node].buffer,
            cl_rx_buf[node].count, NULL) != cl_rx_buf[node].count){
        for (j = 0; j < cl_rx_buf[node].count; j++)
            rte_pktmbuf_free(cl_rx_buf[node].buffer[j]);
        cl->stats.rx_drop += cl_rx_buf[node].count;
    } else
        cl->stats.rx += cl_rx_buf[node].count;

    cl_rx_buf[node].count = 0;
}
```

The second process, the back-end node, receives the packets from the shared ring with the server and send them out, if they belong to the node.

At initialization, it attaches to the server process memory, to have access to the shared ring, parameters and statistics.

```
rx_ring = rte_ring_lookup(get_rx_queue_name(node_id));
if (rx_ring == NULL)
    rte_exit(EXIT_FAILURE, "Cannot get RX ring - "
            "is server process running?\n");

mp = rte_mempool_lookup(PKTMBUF_POOL_NAME);
if (mp == NULL)
    rte_exit(EXIT_FAILURE, "Cannot get mempool for mbufs\n");

mz = rte_memzone_lookup(MZ_SHARED_INFO);
if (mz == NULL)
    rte_exit(EXIT_FAILURE, "Cannot get port info structure\n");
info = mz->addr;
tx_stats = &(info->tx_stats[node_id]);
filter_stats = &(info->filter_stats[node_id]);
```

Then, the hash table that contains the flows that will be handled by the node is created and populated.

```
static struct rte_hash *
create_hash_table(const struct shared_info *info)
{
    uint32_t num_flows_node = info->num_flows / info->num_nodes;
```

(continues on next page)

(continued from previous page)

```
    char name[RTE_HASH_NAMESIZE];
    struct rte_hash *h;

    /* create table */
    struct rte_hash_parameters hash_params = {
        .entries = num_flows_node * 2, /* table load = 50% */
        .key_len = sizeof(uint32_t), /* Store IPv4 dest IP address */
        .socket_id = rte_socket_id(),
        .hash_func_init_val = 0,
    };

    snprintf(name, sizeof(name), "hash_table_%d", node_id);
    hash_params.name = name;
    h = rte_hash_create(&hash_params);

    if (h == NULL)
        rte_exit(EXIT_FAILURE,
                "Problem creating the hash table for node %d\n",
                node_id);
    return h;
}

static void
populate_hash_table(const struct rte_hash *h, const struct shared_info *info)
{
    unsigned int i;
    int32_t ret;
    uint32_t ip_dst;
    uint32_t num_flows_node = 0;
    uint64_t target_node;

    /* Add flows in table */
    for (i = 0; i < info->num_flows; i++) {
        target_node = i % info->num_nodes;
        if (target_node != node_id)
            continue;

        ip_dst = rte_cpu_to_be_32(i);

        ret = rte_hash_add_key(h, (void *) &ip_dst);
        if (ret < 0)
            rte_exit(EXIT_FAILURE, "Unable to add entry %u "
                    "in hash table\n", i);
        else
            num_flows_node++;

    }

    printf("Hash table: Adding 0x%x keys\n", num_flows_node);
}
```

After initialization, packets are dequeued from the shared ring (from the server) and, like in the server process, the IPv4 address from the packets is used as a key to look up in the hash table. If there is a hit, packet is stored in a buffer, to be eventually transmitted in one of the enabled ports. If key is not there, packet is dropped, since the flow is not handled by the node.

```
static inline void
handle_packets(struct rte_hash *h, struct rte_mbuf **bufs, uint16_t num_packets)
{
    struct rte_ipv4_hdr *ipv4_hdr;
```

(continues on next page)

(continued from previous page)

```
    uint32_t ipv4_dst_ip[PKT_READ_SIZE];
    const void *key_ptrs[PKT_READ_SIZE];
    unsigned int i;
    int32_t positions[PKT_READ_SIZE] = {0};

    for (i = 0; i < num_packets; i++) {
        /* Handle IPv4 header.*/
        ipv4_hdr = rte_pktmbuf_mtod_offset(bufs[i], struct rte_ipv4_hdr *,
                sizeof(struct rte_ether_hdr));
        ipv4_dst_ip[i] = ipv4_hdr->dst_addr;
        key_ptrs[i] = &ipv4_dst_ip[i];
    }
    /* Check if packets belongs to any flows handled by this node */
    rte_hash_lookup_bulk(h, key_ptrs, num_packets, positions);

    for (i = 0; i < num_packets; i++) {
        if (likely(positions[i] >= 0)) {
            filter_stats->passed++;
            transmit_packet(bufs[i]);
        } else {
            filter_stats->drop++;
            /* Drop packet, as flow is not handled by this node */
            rte_pktmbuf_free(bufs[i]);
        }
    }
}
```

Finally, note that both processes updates statistics, such as transmitted, received and dropped packets, which are shown and refreshed by the server app.

```
static void
do_stats_display(void)
{
    unsigned int i, j;
    const char clr[] = {27, '[', '2', 'J', '\0'};
    const char topLeft[] = {27, '[', '1', ';', '1', 'H', '\0'};
    uint64_t port_tx[RTE_MAX_ETHPORTS], port_tx_drop[RTE_MAX_ETHPORTS];
    uint64_t node_tx[MAX_NODES], node_tx_drop[MAX_NODES];

    /* to get TX stats, we need to do some summing calculations */
    memset(port_tx, 0, sizeof(port_tx));
    memset(port_tx_drop, 0, sizeof(port_tx_drop));
    memset(node_tx, 0, sizeof(node_tx));
    memset(node_tx_drop, 0, sizeof(node_tx_drop));

    for (i = 0; i < num_nodes; i++) {
        const struct tx_stats *tx = &info->tx_stats[i];

        for (j = 0; j < info->num_ports; j++) {
            const uint64_t tx_val = tx->tx[info->id[j]];
            const uint64_t drop_val = tx->tx_drop[info->id[j]];

            port_tx[j] += tx_val;
            port_tx_drop[j] += drop_val;
            node_tx[i] += tx_val;
            node_tx_drop[i] += drop_val;
        }
    }

    /* Clear screen and move to top left */
```

(continues on next page)

(continued from previous page)

```
    printf("%s%s", clr, topLeft);

    printf("PORTS\n");
    printf("-----\n");
    for (i = 0; i < info->num_ports; i++)
        printf("Port %u: '%s'\t", (unsigned int)info->id[i],
                get_printable_mac_addr(info->id[i]));
    printf("\n\n");
    for (i = 0; i < info->num_ports; i++) {
        printf("Port %u - rx: %9"PRIu64"\t"
                "tx: %9"PRIu64"\n",
                (unsigned int)info->id[i], info->rx_stats.rx[i],
                port_tx[i]);
    }

    printf("\nSERVER\n");
    printf("-----\n");
    printf("distributed: %9"PRIu64", drop: %9"PRIu64"\n",
            flow_dist_stats.distributed, flow_dist_stats.drop);

    printf("\nNODES\n");
    printf("-------\n");
    for (i = 0; i < num_nodes; i++) {
        const unsigned long long rx = nodes[i].stats.rx;
        const unsigned long long rx_drop = nodes[i].stats.rx_drop;
        const struct filter_stats *filter = &info->filter_stats[i];

        printf("Node %2u - rx: %9llu, rx_drop: %9llu\n"
                "            tx: %9"PRIu64", tx_drop: %9"PRIu64"\n"
                "            filter_passed: %9"PRIu64", "
                "filter_drop: %9"PRIu64"\n",
                i, rx, rx_drop, node_tx[i], node_tx_drop[i],
                filter->passed, filter->drop);
    }

    printf("\n");
}
```

4.27 Service Cores Sample Application

The service cores sample application demonstrates the service cores capabilities of DPDK. The service cores infrastructure is part of the DPDK EAL, and allows any DPDK component to register a service. A service is a work item or task, that requires CPU time to perform its duty.

This sample application registers 5 dummy services. These 5 services are used to show how the service_cores API can be used to orchestrate these services to run on different service lcores. This orchestration is done by calling the service cores APIs, however the sample application introduces a "profile" concept to contain the service mapping details. Note that the profile concept is application specific, and not a part of the service cores API.

4.27.1 Compiling the Application

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/service_cores
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linux-gcc
```

   See the *DPDK Getting Started* Guide for possible RTE_TARGET values.

3. Build the application:

```
make
```

4.27.2 Running the Application

To run the example, just execute the binary. Since the application dynamically adds service cores in the application code itself, there is no requirement to pass a service core-mask as an EAL argument at startup time.

```
$ ./build/service_cores
```

4.27.3 Explanation

The following sections provide some explanation of code focusing on registering applications from an applications point of view, and modifying the service core counts and mappings at runtime.

Registering a Service

The following code section shows how to register a service as an application. Note that the service component header must be included by the application in order to register services: `rte_service_component.h`, in addition to the ordinary service cores header `rte_service.h` which provides the runtime functions to add, remove and remap service cores.

```
struct rte_service_spec service = {
        .name = "service_name",
};
int ret = rte_service_component_register(services, &id);
if (ret)
        return -1;

/* set the service itself to be ready to run. In the case of
* ethdev, eventdev etc PMDs, this will be set when the
* appropriate configure or setup function is called.
*/
rte_service_component_runstate_set(id, 1);

/* Collect statistics for the service */
rte_service_set_stats_enable(id, 1);
```

(continues on next page)

(continued from previous page)

```
/* The application sets the service to running state. Note that this
 * function enables the service to run - while the 'component' version
 * of this function (as above) marks the service itself as ready */
ret = rte_service_runstate_set(id, 1);
```

Controlling A Service Core

This section demonstrates how to add a service core. The `rte_service.h` header file provides the functions for dynamically adding and removing cores. The APIs to add and remove cores use lcore IDs similar to existing DPDK functions.

These are the functions to start a service core, and have it run a service:

```
/* the lcore ID to use as a service core */
uint32_t service_core_id = 7;
ret = rte_service_lcore_add(service_core_id);
if(ret)
        return -1;

/* service cores are in "stopped" state when added, so start it */
ret = rte_service_lcore_start(service_core_id);
if(ret)
        return -1;

/* map a service to the service core, causing it to run the service */
uint32_t service_id; /* ID of a registered service */
uint32_t enable = 1; /* 1 maps the service, 0 unmaps */
ret = rte_service_map_lcore_set(service_id, service_core_id, enable);
if(ret)
        return -1;
```

Removing A Service Core

To remove a service core, the steps are similar to adding but in reverse order. Note that it is not allowed to remove a service core if the service is running, and the service-core is the only core running that service (see documentation for `rte_service_lcore_stop` function for details).

Conclusion

The service cores infrastructure provides DPDK with two main features. The first is to abstract away hardware differences: the service core can CPU cycles to a software fallback implementation, allowing the application to be abstracted from the difference in HW / SW availability. The second feature is a flexible method of registering functions to be run, allowing the running of the functions to be scaled across multiple CPUs.

4.28 Multi-process Sample Application

This chapter describes the example applications for multi-processing that are included in the DPDK.

4.28.1 Example Applications

Building the Sample Applications

The multi-process example applications are built in the same way as other sample applications, and as documented in the *DPDK Getting Started Guide*.

To compile the sample application see *Compiling the Sample Applications*.

The applications are located in the `multi_process` sub-directory.

Note: If just a specific multi-process application needs to be built, the final make command can be run just in that application's directory, rather than at the top-level multi-process directory.

Basic Multi-process Example

The examples/simple_mp folder in the DPDK release contains a basic example application to demonstrate how two DPDK processes can work together using queues and memory pools to share information.

Running the Application

To run the application, start one copy of the simple_mp binary in one terminal, passing at least two cores in the coremask/corelist, as follows:

```
./build/simple_mp -l 0-1 -n 4 --proc-type=primary
```

For the first DPDK process run, the proc-type flag can be omitted or set to auto, since all DPDK processes will default to being a primary instance, meaning they have control over the hugepage shared memory regions. The process should start successfully and display a command prompt as follows:

```
$ ./build/simple_mp -l 0-1 -n 4 --proc-type=primary
EAL: coremask set to 3
EAL: Detected lcore 0 on socket 0
EAL: Detected lcore 1 on socket 0
EAL: Detected lcore 2 on socket 0
EAL: Detected lcore 3 on socket 0
...

EAL: Requesting 2 pages of size 1073741824
EAL: Requesting 768 pages of size 2097152
EAL: Ask a virtual area of 0x40000000 bytes
EAL: Virtual area found at 0x7ff200000000 (size = 0x40000000)
...

EAL: check igb_uio module
EAL: check module finished
EAL: Master core 0 is ready (tid=54e41820)
```

(continues on next page)

(continued from previous page)

```
EAL: Core 1 is ready (tid=53b32700)

Starting core 1

simple_mp >
```

To run the secondary process to communicate with the primary process, again run the same binary setting at least two cores in the coremask/corelist:

```
./build/simple_mp -l 2-3 -n 4 --proc-type=secondary
```

When running a secondary process such as that shown above, the proc-type parameter can again be specified as auto. However, omitting the parameter altogether will cause the process to try and start as a primary rather than secondary process.

Once the process type is specified correctly, the process starts up, displaying largely similar status messages to the primary instance as it initializes. Once again, you will be presented with a command prompt.

Once both processes are running, messages can be sent between them using the send command. At any stage, either process can be terminated using the quit command.

```
EAL: Master core 10 is ready (tid=b5f89820)           EAL: Master core 8 is ready␣
↪(tid=864a3820)
EAL: Core 11 is ready (tid=84ffe700)                  EAL: Core 9 is ready (tid=85995700)
Starting core 11                                      Starting core 9
simple_mp > send hello_secondary                      simple_mp > core 9: Received 'hello_
↪secondary'
simple_mp > core 11: Received 'hello_primary'         simple_mp > send hello_primary
simple_mp > quit                                      simple_mp > quit
```

Note: If the primary instance is terminated, the secondary instance must also be shut-down and restarted after the primary. This is necessary because the primary instance will clear and reset the shared memory regions on startup, invalidating the secondary process's pointers. The secondary process can be stopped and restarted without affecting the primary process.

How the Application Works

The core of this example application is based on using two queues and a single memory pool in shared memory. These three objects are created at startup by the primary process, since the secondary process cannot create objects in memory as it cannot reserve memory zones, and the secondary process then uses lookup functions to attach to these objects as it starts up.

```
if (rte_eal_process_type() == RTE_PROC_PRIMARY){
    send_ring = rte_ring_create(_PRI_2_SEC, ring_size, SOCKET0, flags);
    recv_ring = rte_ring_create(_SEC_2_PRI, ring_size, SOCKET0, flags);
    message_pool = rte_mempool_create(_MSG_POOL, pool_size, string_size, pool_cache, priv_data_
↪sz, NULL, NULL, NULL, NULL, SOCKET0, flags);
} else {
    recv_ring = rte_ring_lookup(_PRI_2_SEC);
    send_ring = rte_ring_lookup(_SEC_2_PRI);
    message_pool = rte_mempool_lookup(_MSG_POOL);
}
```

Note, however, that the named ring structure used as send_ring in the primary process is the recv_ring in the secondary process.

Once the rings and memory pools are all available in both the primary and secondary processes, the application simply dedicates two threads to sending and receiving messages respectively. The receive thread simply dequeues any messages on the receive ring, prints them, and frees the buffer space used by the messages back to the memory pool. The send thread makes use of the command-prompt library to interactively request user input for messages to send. Once a send command is issued by the user, a buffer is allocated from the memory pool, filled in with the message contents, then enqueued on the appropriate rte_ring.

Symmetric Multi-process Example

The second example of DPDK multi-process support demonstrates how a set of processes can run in parallel, with each process performing the same set of packet- processing operations. (Since each process is identical in functionality to the others, we refer to this as symmetric multi-processing, to differentiate it from asymmetric multi- processing - such as a client-server mode of operation seen in the next example, where different processes perform different tasks, yet co-operate to form a packet-processing system.) The following diagram shows the data-flow through the application, using two processes.

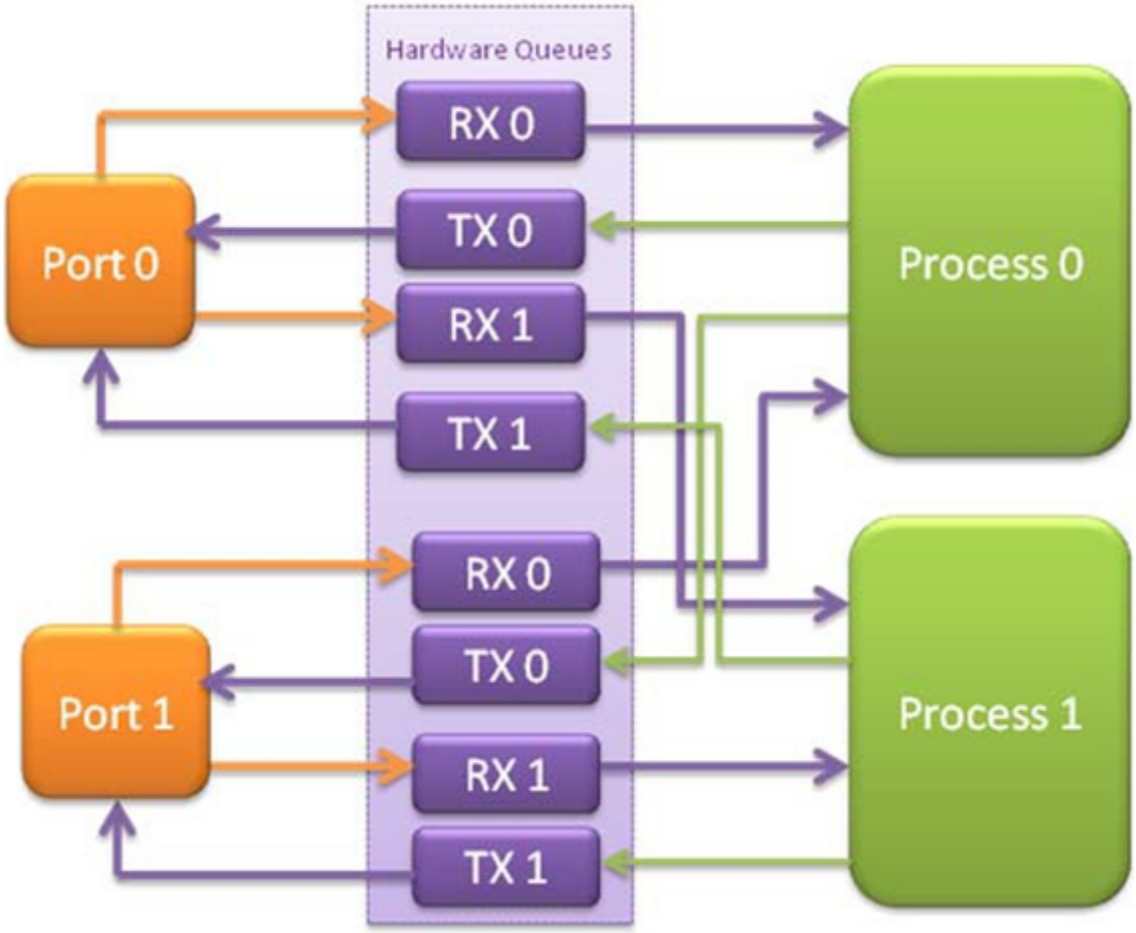


Fig. 4.12: Example Data Flow in a Symmetric Multi-process Application

As the diagram shows, each process reads packets from each of the network ports in use. RSS is used to distribute incoming packets on each port to different hardware RX queues. Each process reads a different

RX queue on each port and so does not contend with any other process for that queue access. Similarly, each process writes outgoing packets to a different TX queue on each port.

Running the Application

As with the simple_mp example, the first instance of the symmetric_mp process must be run as the primary instance, though with a number of other application- specific parameters also provided after the EAL arguments. These additional parameters are:

- -p <portmask>, where portmask is a hexadecimal bitmask of what ports on the system are to be used. For example: -p 3 to use ports 0 and 1 only.
- –num-procs <N>, where N is the total number of symmetric_mp instances that will be run side-by-side to perform packet processing. This parameter is used to configure the appropriate number of receive queues on each network port.
- –proc-id <n>, where n is a numeric value in the range 0 <= n < N (number of processes, specified above). This identifies which symmetric_mp instance is being run, so that each process can read a unique receive queue on each network port.

The secondary symmetric_mp instances must also have these parameters specified, and the first two must be the same as those passed to the primary instance, or errors result.

For example, to run a set of four symmetric_mp instances, running on lcores 1-4, all performing level-2 forwarding of packets between ports 0 and 1, the following commands can be used (assuming run as root):

```
# ./build/symmetric_mp -l 1 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=0
# ./build/symmetric_mp -l 2 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=1
# ./build/symmetric_mp -l 3 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=2
# ./build/symmetric_mp -l 4 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=3
```

Note: In the above example, the process type can be explicitly specified as primary or secondary, rather than auto. When using auto, the first process run creates all the memory structures needed for all processes - irrespective of whether it has a proc-id of 0, 1, 2 or 3.

Note: For the symmetric multi-process example, since all processes work in the same manner, once the hugepage shared memory and the network ports are initialized, it is not necessary to restart all processes if the primary instance dies. Instead, that process can be restarted as a secondary, by explicitly setting the proc-type to secondary on the command line. (All subsequent instances launched will also need this explicitly specified, as auto-detection will detect no primary processes running and therefore attempt to re-initialize shared memory.)

How the Application Works

The initialization calls in both the primary and secondary instances are the same for the most part, calling the rte_eal_init(), 1 G and 10 G driver initialization and then probing devices. Thereafter, the initialization done depends on whether the process is configured as a primary or secondary instance.

In the primary instance, a memory pool is created for the packet mbufs and the network ports to be used are initialized - the number of RX and TX queues per port being determined by the num-procs parameter passed on the command-line. The structures for the initialized network ports are stored in shared memory and therefore will be accessible by the secondary process as it initializes.

```
if (num_ports & 1)
   rte_exit(EXIT_FAILURE, "Application must use an even number of ports\n");

for(i = 0; i < num_ports; i++){
    if(proc_type == RTE_PROC_PRIMARY)
        if (smp_port_init(ports[i], mp, (uint16_t)num_procs) < 0)
            rte_exit(EXIT_FAILURE, "Error initializing ports\n");
}
```

In the secondary instance, rather than initializing the network ports, the port information exported by the primary process is used, giving the secondary process access to the hardware and software rings for each network port. Similarly, the memory pool of mbufs is accessed by doing a lookup for it by name:

```
mp = (proc_type == RTE_PROC_SECONDARY) ? rte_mempool_lookup(_SMP_MBUF_POOL) : rte_mempool_
↪create(_SMP_MBUF_POOL, NB_MBUFS, MBUF_SIZE, ... )
```

Once this initialization is complete, the main loop of each process, both primary and secondary, is exactly the same - each process reads from each port using the queue corresponding to its proc-id parameter, and writes to the corresponding transmit queue on the output port.

Client-Server Multi-process Example

The third example multi-process application included with the DPDK shows how one can use a client-server type multi-process design to do packet processing. In this example, a single server process performs the packet reception from the ports being used and distributes these packets using round-robin ordering among a set of client processes, which perform the actual packet processing. In this case, the client applications just perform level-2 forwarding of packets by sending each packet out on a different network port.

The following diagram shows the data-flow through the application, using two client processes.

Running the Application

The server process must be run initially as the primary process to set up all memory structures for use by the clients. In addition to the EAL parameters, the application- specific parameters are:

- -p <portmask >, where portmask is a hexadecimal bitmask of what ports on the system are to be used. For example: -p 3 to use ports 0 and 1 only.
- -n <num-clients>, where the num-clients parameter is the number of client processes that will process the packets received by the server application.

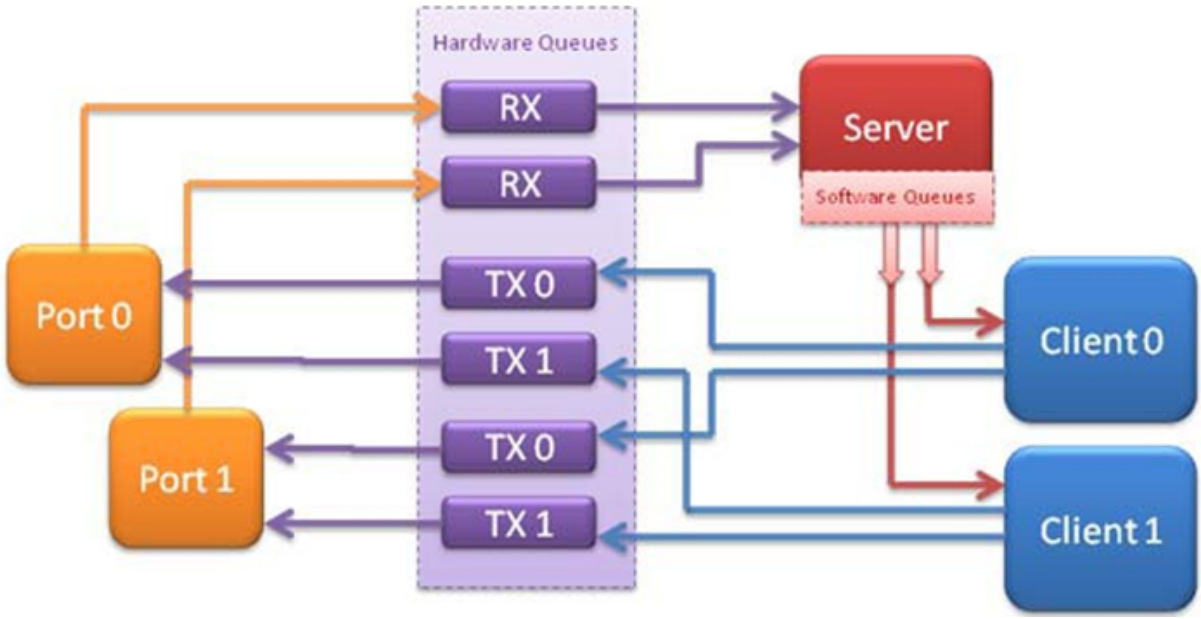


Fig. 4.13: Example Data Flow in a Client-Server Symmetric Multi-process Application

Note: In the server process, a single thread, the master thread, that is, the lowest numbered lcore in the coremask/corelist, performs all packet I/O. If a coremask/corelist is specified with more than a single lcore bit set in it, an additional lcore will be used for a thread to periodically print packet count statistics.

Since the server application stores configuration data in shared memory, including the network ports to be used, the only application parameter needed by a client process is its client instance ID. Therefore, to run a server application on lcore 1 (with lcore 2 printing statistics) along with two client processes running on lcores 3 and 4, the following commands could be used:

```
# ./mp_server/build/mp_server -l 1-2 -n 4 -- -p 3 -n 2
# ./mp_client/build/mp_client -l 3 -n 4 --proc-type=auto -- -n 0
# ./mp_client/build/mp_client -l 4 -n 4 --proc-type=auto -- -n 1
```

Note: If the server application dies and needs to be restarted, all client applications also need to be restarted, as there is no support in the server application for it to run as a secondary process. Any client processes that need restarting can be restarted without affecting the server process.

How the Application Works

The server process performs the network port and data structure initialization much as the symmetric multi-process application does when run as primary. One additional enhancement in this sample application is that the server process stores its port configuration data in a memory zone in hugepage shared memory. This eliminates the need for the client processes to have the portmask parameter passed into them on the command line, as is done for the symmetric multi-process application, and therefore eliminates mismatched parameters as a potential source of errors.

In the same way that the server process is designed to be run as a primary process instance only, the client processes are designed to be run as secondary instances only. They have no code to attempt to create

shared memory objects. Instead, handles to all needed rings and memory pools are obtained via calls to rte_ring_lookup() and rte_mempool_lookup(). The network ports for use by the processes are obtained by loading the network port drivers and probing the PCI bus, which will, as in the symmetric multi-process example, automatically get access to the network ports using the settings already configured by the primary/server process.

Once all applications are initialized, the server operates by reading packets from each network port in turn and distributing those packets to the client queues (software rings, one for each client process) in round-robin order. On the client side, the packets are read from the rings in as big of bursts as possible, then routed out to a different network port. The routing used is very simple. All packets received on the first NIC port are transmitted back out on the second port and vice versa. Similarly, packets are routed between the 3rd and 4th network ports and so on. The sending of packets is done by writing the packets directly to the network ports; they are not transferred back via the server process.

In both the server and the client processes, outgoing packets are buffered before being sent, so as to allow the sending of multiple packets in a single burst to improve efficiency. For example, the client process will buffer packets to send, until either the buffer is full or until we receive no further packets from the server.

4.29 QoS Metering Sample Application

The QoS meter sample application is an example that demonstrates the use of DPDK to provide QoS marking and metering, as defined by RFC2697 for Single Rate Three Color Marker (srTCM) and RFC 2698 for Two Rate Three Color Marker (trTCM) algorithm.

4.29.1 Overview

The application uses a single thread for reading the packets from the RX port, metering, marking them with the appropriate color (green, yellow or red) and writing them to the TX port.

A policing scheme can be applied before writing the packets to the TX port by dropping or changing the color of the packet in a static manner depending on both the input and output colors of the packets that are processed by the meter.

The operation mode can be selected as compile time out of the following options:

- Simple forwarding
- srTCM color blind
- srTCM color aware
- srTCM color blind
- srTCM color aware

Please refer to RFC2697 and RFC2698 for details about the srTCM and trTCM configurable parameters (CIR, CBS and EBS for srTCM; CIR, PIR, CBS and PBS for trTCM).

The color blind modes are functionally equivalent with the color-aware modes when all the incoming packets are colored as green.

4.29.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `qos_meter` sub-directory.

4.29.3 Running the Application

The application execution command line is as below:

```
./qos_meter [EAL options] -- -p PORTMASK
```

The application is constrained to use a single core in the EAL core mask and 2 ports only in the application port mask (first port from the port mask is used for RX and the other port in the core mask is used for TX).

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.29.4 Explanation

Selecting one of the metering modes is done with these defines:

```
#define APP_MODE_FWD    0
#define APP_MODE_SRTCM_COLOR_BLIND  1
#define APP_MODE_SRTCM_COLOR_AWARE  2
#define APP_MODE_TRTCM_COLOR_BLIND  3
#define APP_MODE_TRTCM_COLOR_AWARE  4

#define APP_MODE  APP_MODE_SRTCM_COLOR_BLIND
```

To simplify debugging (for example, by using the traffic generator RX side MAC address based packet filtering feature), the color is defined as the LSB byte of the destination MAC address.

The traffic meter parameters are configured in the application source code with following default values:

```
struct rte_meter_srtcm_params app_srtcm_params[] = {

    {.cir = 1000000 * 46, .cbs = 2048, .ebs = 2048},

};

struct rte_meter_trtcm_params app_trtcm_params[] = {

    {.cir = 1000000 * 46, .pir = 1500000 * 46, .cbs = 2048, .pbs = 2048},

};
```

Assuming the input traffic is generated at line rate and all packets are 64 bytes Ethernet frames (IPv4 packet size of 46 bytes) and green, the expected output traffic should be marked as shown in the following table:

Table 4.1: Output Traffic Marking

Mode	Green (Mpps)	Yellow (Mpps)	Red (Mpps)
srTCM blind	1	1	12.88
srTCM color	1	1	12.88
trTCM blind	1	0.5	13.38
trTCM color	1	0.5	13.38
FWD	14.88	0	0

To set up the policing scheme as desired, it is necessary to modify the main.h source file, where this policy is implemented as a static structure, as follows:

```
int policer_table[e_RTE_METER_COLORS][e_RTE_METER_COLORS] =
{
   { GREEN, RED, RED},
   { DROP, YELLOW, RED},
   { DROP, DROP, RED}
};
```

Where rows indicate the input color, columns indicate the output color, and the value that is stored in the table indicates the action to be taken for that particular case.

There are four different actions:

- GREEN: The packet's color is changed to green.
- YELLOW: The packet's color is changed to yellow.
- RED: The packet's color is changed to red.
- DROP: The packet is dropped.

In this particular case:

- Every packet which input and output color are the same, keeps the same color.
- Every packet which color has improved is dropped (this particular case can't happen, so these values will not be used).
- For the rest of the cases, the color is changed to red.

Note:

- In color blind mode, first row GREEN color is only valid.
- To drop the packet, policer_table action has to be set to DROP.

4.30 QoS Scheduler Sample Application

The QoS sample application demonstrates the use of the DPDK to provide QoS scheduling.

4.30.1 Overview

The architecture of the QoS scheduler application is shown in the following figure.

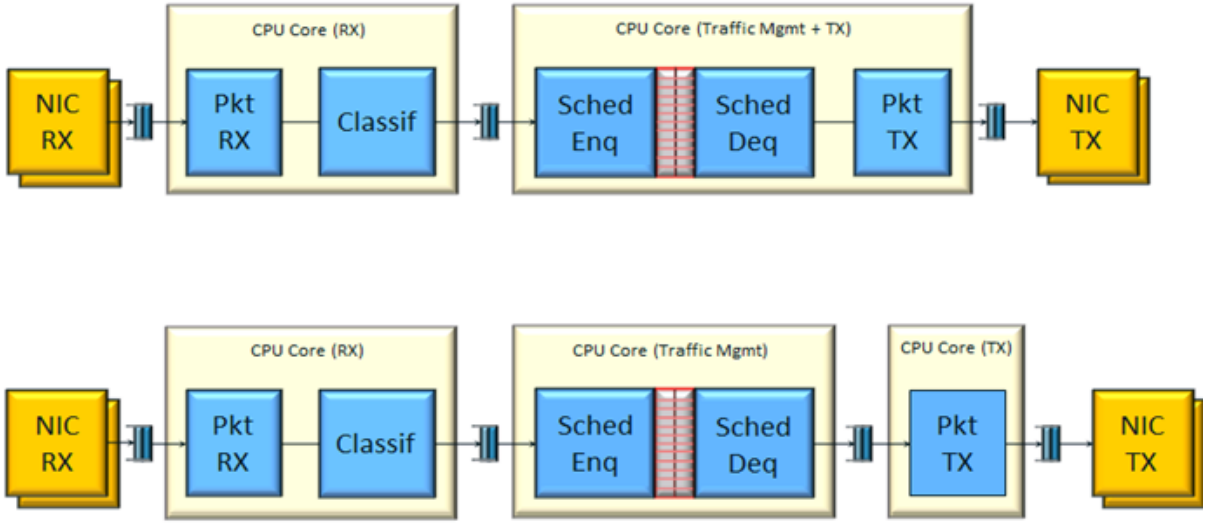


Fig. 4.14: QoS Scheduler Application Architecture

There are two flavors of the runtime execution for this application, with two or three threads per each packet flow configuration being used. The RX thread reads packets from the RX port, classifies the packets based on the double VLAN (outer and inner) and the lower byte of the IP destination address and puts them into the ring queue. The worker thread dequeues the packets from the ring and calls the QoS scheduler enqueue/dequeue functions. If a separate TX core is used, these are sent to the TX ring. Otherwise, they are sent directly to the TX port. The TX thread, if present, reads from the TX ring and write the packets to the TX port.

4.30.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `qos_sched` sub-directory.

> **Note:** This application is intended as a linux only.

Note: To get statistics on the sample app using the command line interface as described in the next section, DPDK must be compiled defining *CONFIG_RTE_SCHED_COLLECT_STATS*, which can be done by changing the configuration file for the specific target to be compiled.

4.30.3 Running the Application

Note: In order to run the application, a total of at least 4 G of huge pages must be set up for each of the used sockets (depending on the cores in use).

The application has a number of command line options:

```
./qos_sched [EAL options] -- <APP PARAMS>
```

Mandatory application parameters include:

- –pfc "RX PORT, TX PORT, RX LCORE, WT LCORE, TX CORE": Packet flow configuration. Multiple pfc entities can be configured in the command line, having 4 or 5 items (if TX core defined or not).

Optional application parameters include:

- -i: It makes the application to start in the interactive mode. In this mode, the application shows a command line that can be used for obtaining statistics while scheduling is taking place (see interactive mode below for more information).
- –mst n: Master core index (the default value is 1).
- –rsz "A, B, C": Ring sizes:
- A = Size (in number of buffer descriptors) of each of the NIC RX rings read by the I/O RX lcores (the default value is 128).
- B = Size (in number of elements) of each of the software rings used by the I/O RX lcores to send packets to worker lcores (the default value is 8192).
- C = Size (in number of buffer descriptors) of each of the NIC TX rings written by worker lcores (the default value is 256)
- –bsz "A, B, C, D": Burst sizes
- A = I/O RX lcore read burst size from the NIC RX (the default value is 64)
- B = I/O RX lcore write burst size to the output software rings, worker lcore read burst size from input software rings,QoS enqueue size (the default value is 64)
- C = QoS dequeue size (the default value is 32)
- D = Worker lcore write burst size to the NIC TX (the default value is 64)
- –msz M: Mempool size (in number of mbufs) for each pfc (default 2097152)
- –rth "A, B, C": The RX queue threshold parameters
- A = RX prefetch threshold (the default value is 8)
- B = RX host threshold (the default value is 8)
- C = RX write-back threshold (the default value is 4)
- –tth "A, B, C": TX queue threshold parameters
- A = TX prefetch threshold (the default value is 36)
- B = TX host threshold (the default value is 0)

- C = TX write-back threshold (the default value is 0)
- –cfg FILE: Profile configuration to load

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The profile configuration file defines all the port/subport/pipe/traffic class/queue parameters needed for the QoS scheduler configuration.

The profile file has the following format:

```
; port configuration [port]

frame overhead = 24
number of subports per port = 1

; Subport configuration

[subport 0]
number of pipes per subport = 4096
queue sizes = 64 64 64 64 64 64 64 64 64 64 64 64 64
tb rate = 1250000000; Bytes per second
tb size = 1000000; Bytes
tc 0 rate = 1250000000;     Bytes per second
tc 1 rate = 1250000000;     Bytes per second
tc 2 rate = 1250000000;     Bytes per second
tc 3 rate = 1250000000;     Bytes per second
tc 4 rate = 1250000000;     Bytes per second
tc 5 rate = 1250000000;     Bytes per second
tc 6 rate = 1250000000;     Bytes per second
tc 7 rate = 1250000000;     Bytes per second
tc 8 rate = 1250000000;     Bytes per second
tc 9 rate = 1250000000;     Bytes per second
tc 10 rate = 1250000000;     Bytes per second
tc 11 rate = 1250000000;     Bytes per second
tc 12 rate = 1250000000;     Bytes per second

tc period = 10;             Milliseconds
tc oversubscription period = 10;     Milliseconds

pipe 0-4095 = 0;        These pipes are configured with pipe profile 0

; Pipe configuration

[pipe profile 0]
tb rate = 305175; Bytes per second
tb size = 1000000; Bytes

tc 0 rate = 305175; Bytes per second
tc 1 rate = 305175; Bytes per second
tc 2 rate = 305175; Bytes per second
tc 3 rate = 305175; Bytes per second
tc 4 rate = 305175; Bytes per second
tc 5 rate = 305175; Bytes per second
tc 6 rate = 305175; Bytes per second
tc 7 rate = 305175; Bytes per second
tc 8 rate = 305175; Bytes per second
tc 9 rate = 305175; Bytes per second
tc 10 rate = 305175; Bytes per second
tc 11 rate = 305175; Bytes per second
tc 12 rate = 305175; Bytes per second
```

(continues on next page)

(continued from previous page)

```
tc period = 40; Milliseconds

tc 0 oversubscription weight = 1
tc 1 oversubscription weight = 1
tc 2 oversubscription weight = 1
tc 3 oversubscription weight = 1
tc 4 oversubscription weight = 1
tc 5 oversubscription weight = 1
tc 6 oversubscription weight = 1
tc 7 oversubscription weight = 1
tc 8 oversubscription weight = 1
tc 9 oversubscription weight = 1
tc 10 oversubscription weight = 1
tc 11 oversubscription weight = 1
tc 12 oversubscription weight = 1

tc 12 wrr weights = 1 1 1 1

; RED params per traffic class and color (Green / Yellow / Red)

[red]
tc 0 wred min = 48 40 32
tc 0 wred max = 64 64 64
tc 0 wred inv prob = 10 10 10
tc 0 wred weight = 9 9 9

tc 1 wred min = 48 40 32
tc 1 wred max = 64 64 64
tc 1 wred inv prob = 10 10 10
tc 1 wred weight = 9 9 9

tc 2 wred min = 48 40 32
tc 2 wred max = 64 64 64
tc 2 wred inv prob = 10 10 10
tc 2 wred weight = 9 9 9

tc 3 wred min = 48 40 32
tc 3 wred max = 64 64 64
tc 3 wred inv prob = 10 10 10
tc 3 wred weight = 9 9 9

tc 4 wred min = 48 40 32
tc 4 wred max = 64 64 64
tc 4 wred inv prob = 10 10 10
tc 4 wred weight = 9 9 9

tc 5 wred min = 48 40 32
tc 5 wred max = 64 64 64
tc 5 wred inv prob = 10 10 10
tc 5 wred weight = 9 9 9

tc 6 wred min = 48 40 32
tc 6 wred max = 64 64 64
tc 6 wred inv prob = 10 10 10
tc 6 wred weight = 9 9 9

tc 7 wred min = 48 40 32
tc 7 wred max = 64 64 64
tc 7 wred inv prob = 10 10 10
tc 7 wred weight = 9 9 9
```

(continues on next page)

(continued from previous page)

```
tc 8 wred min = 48 40 32
tc 8 wred max = 64 64 64
tc 8 wred inv prob = 10 10 10
tc 8 wred weight = 9 9 9

tc 9 wred min = 48 40 32
tc 9 wred max = 64 64 64
tc 9 wred inv prob = 10 10 10
tc 9 wred weight = 9 9 9

tc 10 wred min = 48 40 32
tc 10 wred max = 64 64 64
tc 10 wred inv prob = 10 10 10
tc 10 wred weight = 9 9 9

tc 11 wred min = 48 40 32
tc 11 wred max = 64 64 64
tc 11 wred inv prob = 10 10 10
tc 11 wred weight = 9 9 9

tc 12 wred min = 48 40 32
tc 12 wred max = 64 64 64
tc 12 wred inv prob = 10 10 10
tc 12 wred weight = 9 9 9
```

Interactive mode

These are the commands that are currently working under the command line interface:

- Control Commands
- –quit: Quits the application.
- General Statistics
  - stats app: Shows a table with in-app calculated statistics.
  - stats port X subport Y: For a specific subport, it shows the number of packets that went through the scheduler properly and the number of packets that were dropped. The same information is shown in bytes. The information is displayed in a table separating it in different traffic classes.
  - stats port X subport Y pipe Z: For a specific pipe, it shows the number of packets that went through the scheduler properly and the number of packets that were dropped. The same information is shown in bytes. This information is displayed in a table separating it in individual queues.
- Average queue size

All of these commands work the same way, averaging the number of packets throughout a specific subset of queues.

Two parameters can be configured for this prior to calling any of these commands:

- qavg n X: n is the number of times that the calculation will take place. Bigger numbers provide higher accuracy. The default value is 10.
- qavg period X: period is the number of microseconds that will be allowed between each calculation. The default value is 100.

The commands that can be used for measuring average queue size are:

- qavg port X subport Y: Show average queue size per subport.
- qavg port X subport Y tc Z: Show average queue size per subport for a specific traffic class.
- qavg port X subport Y pipe Z: Show average queue size per pipe.
- qavg port X subport Y pipe Z tc A: Show average queue size per pipe for a specific traffic class.
- qavg port X subport Y pipe Z tc A q B: Show average queue size of a specific queue.

Example

The following is an example command with a single packet flow configuration:

```
./qos_sched -l 1,5,7 -n 4 -- --pfc "3,2,5,7" --cfg ./profile.cfg
```

This example uses a single packet flow configuration which creates one RX thread on lcore 5 reading from port 3 and a worker thread on lcore 7 writing to port 2.

Another example with 2 packet flow configurations using different ports but sharing the same core for QoS scheduler is given below:

```
./qos_sched -l 1,2,6,7 -n 4 -- --pfc "3,2,2,6,7" --pfc "1,0,2,6,7" --cfg ./profile.cfg
```

Note that independent cores for the packet flow configurations for each of the RX, WT and TX thread are also supported, providing flexibility to balance the work.

The EAL coremask/corelist is constrained to contain the default mastercore 1 and the RX, WT and TX cores only.

4.30.4 Explanation

The Port/Subport/Pipe/Traffic Class/Queue are the hierarchical entities in a typical QoS application:

- A subport represents a predefined group of users.
- A pipe represents an individual user/subscriber.
- A traffic class is the representation of a different traffic type with a specific loss rate, delay and jitter requirements; such as data voice, video or data transfers.
- A queue hosts packets from one or multiple connections of the same type belonging to the same user.

The traffic flows that need to be configured are application dependent. This application classifies based on the QinQ double VLAN tags and the IP destination address as indicated in the following table.

Table 4.2: Entity Types

Level Name	Siblings per Parent	QoS Functional Description	Selected By
Port	•	Ethernet port	Physical port
Subport	Config (8)	Traffic shaped (token bucket)	Outer VLAN tag
Pipe	Config (4k)	Traffic shaped (token bucket)	Inner VLAN tag
Traffic Class	13	TCs of the same pipe services in strict priority	Destination IP address (0.0.0.X)
Queue	High Priority TC: 1, Lowest Priority TC: 4	Queue of lowest priority traffic class (Best effort) serviced in WRR	Destination IP address (0.0.0.X)

Please refer to the "QoS Scheduler" chapter in the *DPDK Programmer's Guide* for more information about these parameters.

4.31 Timer Sample Application

The Timer sample application is a simple application that demonstrates the use of a timer in a DPDK application. This application prints some messages from different lcores regularly, demonstrating the use of timers.

4.31.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `timer` sub-directory.

4.31.2 Running the Application

To run the example in linux environment:

```
$ ./build/timer -l 0-3 -n 4
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.31.3 Explanation

The following sections provide some explanation of the code.

Initialization and Main Loop

In addition to EAL initialization, the timer subsystem must be initialized, by calling the rte_timer_subsystem_init() function.

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_panic("Cannot init EAL\n");

/* init RTE timer library */

rte_timer_subsystem_init();
```

After timer creation (see the next paragraph), the main loop is executed on each slave lcore using the well-known rte_eal_remote_launch() and also on the master.

```
/* call lcore_mainloop() on every slave lcore  */

RTE_LCORE_FOREACH_SLAVE(lcore_id) {
    rte_eal_remote_launch(lcore_mainloop, NULL, lcore_id);
}

/* call it on master lcore too */

(void) lcore_mainloop(NULL);
```

The main loop is very simple in this example:

```
while (1) {
    /*
     *   Call the timer handler on each core: as we don't
     *   need a very precise timer, so only call
     *   rte_timer_manage() every ~10ms (at 2 GHz). In a real
     *   application, this will enhance performances as
     *   reading the HPET timer is not efficient.
    */

    cur_tsc = rte_rdtsc();

    diff_tsc = cur_tsc - prev_tsc;

    if (diff_tsc > TIMER_RESOLUTION_CYCLES) {
        rte_timer_manage();
        prev_tsc = cur_tsc;
    }
}
```

As explained in the comment, it is better to use the TSC register (as it is a per-lcore register) to check if the rte_timer_manage() function must be called or not. In this example, the resolution of the timer is 10 milliseconds.

Managing Timers

In the main() function, the two timers are initialized. This call to rte_timer_init() is necessary before doing any other operation on the timer structure.

```
/* init timer structures */

rte_timer_init(&timer0);
rte_timer_init(&timer1);
```

Then, the two timers are configured:

- The first timer (timer0) is loaded on the master lcore and expires every second. Since the PERIODICAL flag is provided, the timer is reloaded automatically by the timer subsystem. The callback function is timer0_cb().
- The second timer (timer1) is loaded on the next available lcore every 333 ms. The SINGLE flag means that the timer expires only once and must be reloaded manually if required. The callback function is timer1_cb().

```
/* load timer0, every second, on master lcore, reloaded automatically */

hz = rte_get_hpet_hz();

lcore_id = rte_lcore_id();

rte_timer_reset(&timer0, hz, PERIODICAL, lcore_id, timer0_cb, NULL);

/* load timer1, every second/3, on next lcore, reloaded manually */

lcore_id = rte_get_next_lcore(lcore_id, 0, 1);

rte_timer_reset(&timer1, hz/3, SINGLE, lcore_id, timer1_cb, NULL);
```

The callback for the first timer (timer0) only displays a message until a global counter reaches 20 (after 20 seconds). In this case, the timer is stopped using the rte_timer_stop() function.

```
/* timer0 callback */

static void
timer0_cb(__rte_unused struct rte_timer *tim, __rte_unused void *arg)
{
    static unsigned counter = 0;

    unsigned lcore_id = rte_lcore_id();

    printf("%s() on lcore %u\n", FUNCTION , lcore_id);

    /* this timer is automatically reloaded until we decide to stop it, when counter reaches␣
↪20. */

    if ((counter ++) == 20)
        rte_timer_stop(tim);
}
```

The callback for the second timer (timer1) displays a message and reloads the timer on the next lcore, using the rte_timer_reset() function:

```
/* timer1 callback */

static void
timer1_cb(__rte_unused struct rte_timer *tim, __rte_unused void *arg)
{
    unsigned lcore_id = rte_lcore_id();
    uint64_t hz;

    printf("%s() on lcore %u\\n", FUNCTION , lcore_id);

    /* reload it on another lcore */

    hz = rte_get_hpet_hz();

    lcore_id = rte_get_next_lcore(lcore_id, 0, 1);

    rte_timer_reset(&timer1, hz/3, SINGLE, lcore_id, timer1_cb, NULL);
}
```

4.32 Packet Ordering Application

The Packet Ordering sample app simply shows the impact of reordering a stream. It's meant to stress the library with different configurations for performance.

4.32.1 Overview

The application uses at least three CPU cores:

- RX core (maser core) receives traffic from the NIC ports and feeds Worker cores with traffic through SW queues.
- Worker core (slave core) basically do some light work on the packet. Currently it modifies the output port of the packet for configurations with more than one port enabled.
- TX Core (slave core) receives traffic from Worker cores through software queues, inserts out-of-order packets into reorder buffer, extracts ordered packets from the reorder buffer and sends them to the NIC ports for transmission.

4.32.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `packet_ordering` sub-directory.

4.32.3 Running the Application

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

Application Command Line

The application execution command line is:

```
./packet_ordering [EAL options] -- -p PORTMASK [--disable-reorder] [--insight-worker]
```

The -c EAL CPU_COREMASK option has to contain at least 3 CPU cores. The first CPU core in the core mask is the master core and would be assigned to RX core, the last to TX core and the rest to Worker cores.

The PORTMASK parameter must contain either 1 or even enabled port numbers. When setting more than 1 port, traffic would be forwarded in pairs. For example, if we enable 4 ports, traffic from port 0 to 1 and from 1 to 0, then the other pair from 2 to 3 and from 3 to 2, having [0,1] and [2,3] pairs.

The disable-reorder long option does, as its name implies, disable the reordering of traffic, which should help evaluate reordering performance impact.

The insight-worker long option enables output the packet statistics of each worker thread.

4.33 VMDQ and DCB Forwarding Sample Application

The VMDQ and DCB Forwarding sample application is a simple example of packet processing using the DPDK. The application performs L2 forwarding using VMDQ and DCB to divide the incoming traffic into queues. The traffic splitting is performed in hardware by the VMDQ and DCB features of the Intel® 82599 and X710/XL710 Ethernet Controllers.

4.33.1 Overview

This sample application can be used as a starting point for developing a new application that is based on the DPDK and uses VMDQ and DCB for traffic partitioning.

The VMDQ and DCB filters work on MAC and VLAN traffic to divide the traffic into input queues on the basis of the Destination MAC address, VLAN ID and VLAN user priority fields. VMDQ filters split the traffic into 16 or 32 groups based on the Destination MAC and VLAN ID. Then, DCB places each packet into one of queues within that group, based upon the VLAN user priority field.

All traffic is read from a single incoming port (port 0) and output on port 1, without any processing being performed. With Intel® 82599 NIC, for example, the traffic is split into 128 queues on input, where each thread of the application reads from multiple queues. When run with 8 threads, that is, with the -c FF option, each thread receives and forwards packets from 16 queues.

As supplied, the sample application configures the VMDQ feature to have 32 pools with 4 queues each as indicated in Fig. 4.15. The Intel® 82599 10 Gigabit Ethernet Controller NIC also supports the splitting of traffic into 16 pools of 8 queues. While the Intel® X710 or XL710 Ethernet Controller NICs support many configurations of VMDQ pools of 4 or 8 queues each. For simplicity, only 16 or 32 pools

is supported in this sample. And queues numbers for each VMDQ pool can be changed by setting CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_VM in config/common_* file. The nb-pools, nb-tcs and enable-rss parameters can be passed on the command line, after the EAL parameters:

```
./build/vmdq_dcb [EAL options] -- -p PORTMASK --nb-pools NP --nb-tcs TC --enable-rss
```

where, NP can be 16 or 32, TC can be 4 or 8, rss is disabled by default.

Fig. 4.15: Packet Flow Through the VMDQ and DCB Sample Application

In Linux* user space, the application can display statistics with the number of packets received on each queue. To have the application display the statistics, send a SIGHUP signal to the running application process.

The VMDQ and DCB Forwarding sample application is in many ways simpler than the L2 Forwarding application (see *L2 Forwarding Sample Application (in Real and Virtualized Environments)*) as it performs unidirectional L2 forwarding of packets from one port to a second port. No command-line options are taken by this application apart from the standard EAL command-line options.

Note: Since VMD queues are being used for VMM, this application works correctly when VTd is disabled in the BIOS or Linux* kernel (intel_iommu=off).

4.33.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `vmdq_dcb` sub-directory.

4.33.3 Running the Application

To run the example in a linux environment:

```
user@target:~$ ./build/vmdq_dcb -l 0-3 -n 4 -- -p 0x3 --nb-pools 32 --nb-tcs 4
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.33.4 Explanation

The following sections provide some explanation of the code.

Initialization

The EAL, driver and PCI configuration is performed largely as in the L2 Forwarding sample application, as is the creation of the mbuf pool. See *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. Where this example application differs is in the configuration of the NIC port for RX.

The VMDQ and DCB hardware feature is configured at port initialization time by setting the appropriate values in the rte_eth_conf structure passed to the rte_eth_dev_configure() API. Initially in the application, a default structure is provided for VMDQ and DCB configuration to be filled in later by the application.

```
/* empty vmdq+dcb configuration structure. Filled in programmatically */
static const struct rte_eth_conf vmdq_dcb_conf_default = {
    .rxmode = {
        .mq_mode        = ETH_MQ_RX_VMDQ_DCB,
        .split_hdr_size = 0,
    },
    .txmode = {
        .mq_mode = ETH_MQ_TX_VMDQ_DCB,
    },
    /*
     * should be overridden separately in code with
     * appropriate values
     */
    .rx_adv_conf = {
        .vmdq_dcb_conf = {
            .nb_queue_pools = ETH_32_POOLS,
            .enable_default_pool = 0,
            .default_pool = 0,
            .nb_pool_maps = 0,
            .pool_map = {{0, 0},},
            .dcb_tc = {0},
        },
        .dcb_rx_conf = {
            .nb_tcs = ETH_4_TCS,
            /** Traffic class each UP mapped to. */
            .dcb_tc = {0},
        },
        .vmdq_rx_conf = {
            .nb_queue_pools = ETH_32_POOLS,
            .enable_default_pool = 0,
            .default_pool = 0,
            .nb_pool_maps = 0,
            .pool_map = {{0, 0},},
        },
    },
    .tx_adv_conf = {
        .vmdq_dcb_tx_conf = {
            .nb_queue_pools = ETH_32_POOLS,
            .dcb_tc = {0},
        },
    },
};
```

The get_eth_conf() function fills in an rte_eth_conf structure with the appropriate values, based on the global vlan_tags array, and dividing up the possible user priority values equally among the individual queues (also referred to as traffic classes) within each pool. With Intel® 82599 NIC, if the number of pools is 32, then the user priority fields are allocated 2 to a queue. If 16 pools are used, then each of the 8 user priority fields is allocated to its own queue within the pool. With Intel® X710/XL710 NICs, if number of tcs is 4, and number of queues in pool is 8, then the user priority fields are allocated 2 to one tc, and a tc has 2 queues mapping to it, then RSS will determine the destination queue in 2. For the

VLAN IDs, each one can be allocated to possibly multiple pools of queues, so the pools parameter in the rte_eth_vmdq_dcb_conf structure is specified as a bitmask value. For destination MAC, each VMDQ pool will be assigned with a MAC address. In this sample, each VMDQ pool is assigned to the MAC like 52:54:00:12:<port_id>:<pool_id>, that is, the MAC of VMDQ pool 2 on port 1 is 52:54:00:12:01:02.

```
const uint16_t vlan_tags[] = {
    0, 1, 2, 3, 4, 5, 6, 7,
    8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, 23,
    24, 25, 26, 27, 28, 29, 30, 31
};

/* pool mac addr template, pool mac addr is like: 52 54 00 12 port# pool# */
static struct rte_ether_addr pool_addr_template = {
    .addr_bytes = {0x52, 0x54, 0x00, 0x12, 0x00, 0x00}
};

/* Builds up the correct configuration for vmdq+dcb based on the vlan tags array
 * given above, and the number of traffic classes available for use. */
static inline int
get_eth_conf(struct rte_eth_conf *eth_conf)
{
    struct rte_eth_vmdq_dcb_conf conf;
    struct rte_eth_vmdq_rx_conf  vmdq_conf;
    struct rte_eth_dcb_rx_conf   dcb_conf;
    struct rte_eth_vmdq_dcb_tx_conf tx_conf;
    uint8_t i;

    conf.nb_queue_pools = (enum rte_eth_nb_pools)num_pools;
    vmdq_conf.nb_queue_pools = (enum rte_eth_nb_pools)num_pools;
    tx_conf.nb_queue_pools = (enum rte_eth_nb_pools)num_pools;
    conf.nb_pool_maps = num_pools;
    vmdq_conf.nb_pool_maps = num_pools;
    conf.enable_default_pool = 0;
    vmdq_conf.enable_default_pool = 0;
    conf.default_pool = 0; /* set explicit value, even if not used */
    vmdq_conf.default_pool = 0;

    for (i = 0; i < conf.nb_pool_maps; i++) {
        conf.pool_map[i].vlan_id = vlan_tags[i];
        vmdq_conf.pool_map[i].vlan_id = vlan_tags[i];
        conf.pool_map[i].pools = 1UL << i ;
        vmdq_conf.pool_map[i].pools = 1UL << i;
    }
    for (i = 0; i < ETH_DCB_NUM_USER_PRIORITIES; i++){
        conf.dcb_tc[i] = i % num_tcs;
        dcb_conf.dcb_tc[i] = i % num_tcs;
        tx_conf.dcb_tc[i] = i % num_tcs;
    }
    dcb_conf.nb_tcs = (enum rte_eth_nb_tcs)num_tcs;
    (void)(rte_memcpy(eth_conf, &vmdq_dcb_conf_default, sizeof(*eth_conf)));
    (void)(rte_memcpy(&eth_conf->rx_adv_conf.vmdq_dcb_conf, &conf,
               sizeof(conf)));
    (void)(rte_memcpy(&eth_conf->rx_adv_conf.dcb_rx_conf, &dcb_conf,
               sizeof(dcb_conf)));
    (void)(rte_memcpy(&eth_conf->rx_adv_conf.vmdq_rx_conf, &vmdq_conf,
               sizeof(vmdq_conf)));
    (void)(rte_memcpy(&eth_conf->tx_adv_conf.vmdq_dcb_tx_conf, &tx_conf,
               sizeof(tx_conf)));
    if (rss_enable) {
        eth_conf->rxmode.mq_mode= ETH_MQ_RX_VMDQ_DCB_RSS;
        eth_conf->rx_adv_conf.rss_conf.rss_hf = ETH_RSS_IP |
```

(continues on next page)

(continued from previous page)

```
                            ETH_RSS_UDP |
                            ETH_RSS_TCP |
                            ETH_RSS_SCTP;
    }
    return 0;
}

......

/* Set mac for each pool.*/
for (q = 0; q < num_pools; q++) {
    struct rte_ether_addr mac;
    mac = pool_addr_template;
    mac.addr_bytes[4] = port;
    mac.addr_bytes[5] = q;
    printf("Port %u vmdq pool %u set mac %02x:%02x:%02x:%02x:%02x:%02x\n",
        port, q,
        mac.addr_bytes[0], mac.addr_bytes[1],
        mac.addr_bytes[2], mac.addr_bytes[3],
        mac.addr_bytes[4], mac.addr_bytes[5]);
    retval = rte_eth_dev_mac_addr_add(port, &mac,
            q + vmdq_pool_base);
    if (retval) {
        printf("mac addr add failed at pool %d\n", q);
        return retval;
    }
}
```

Once the network port has been initialized using the correct VMDQ and DCB values, the initialization of the port's RX and TX hardware rings is performed similarly to that in the L2 Forwarding sample application. See *L2 Forwarding Sample Application (in Real and Virtualized Environments)* for more information.

Statistics Display

When run in a linux environment, the VMDQ and DCB Forwarding sample application can display statistics showing the number of packets read from each RX queue. This is provided by way of a signal handler for the SIGHUP signal, which simply prints to standard output the packet counts in grid form. Each row of the output is a single pool with the columns being the queue number within that pool.

To generate the statistics output, use the following command:

```
user@host$ sudo killall -HUP vmdq_dcb_app
```

Please note that the statistics output will appear on the terminal where the vmdq_dcb_app is running, rather than the terminal from which the HUP signal was sent.

4.34 VMDq Forwarding Sample Application

The VMDq Forwarding sample application is a simple example of packet processing using the DPDK. The application performs L2 forwarding using VMDq to divide the incoming traffic into queues. The traffic splitting is performed in hardware by the VMDq feature of the Intel® 82599 and X710/XL710 Ethernet Controllers.

4.34.1 Overview

This sample application can be used as a starting point for developing a new application that is based on the DPDK and uses VMDq for traffic partitioning.

VMDq filters split the incoming packets up into different "pools" - each with its own set of RX queues - based upon the MAC address and VLAN ID within the VLAN tag of the packet.

All traffic is read from a single incoming port and output on another port, without any processing being performed. With Intel® 82599 NIC, for example, the traffic is split into 128 queues on input, where each thread of the application reads from multiple queues. When run with 8 threads, that is, with the -c FF option, each thread receives and forwards packets from 16 queues.

As supplied, the sample application configures the VMDq feature to have 32 pools with 4 queues each. The Intel® 82599 10 Gigabit Ethernet Controller NIC also supports the splitting of traffic into 16 pools of 2 queues. While the Intel® X710 or XL710 Ethernet Controller NICs support many configurations of VMDq pools of 4 or 8 queues each. And queues numbers for each VMDq pool can be changed by setting CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_VM in config/common_* file. The nb-pools and enable-rss parameters can be passed on the command line, after the EAL parameters:

```
./build/vmdq_app [EAL options] -- -p PORTMASK --nb-pools NP --enable-rss
```

where, NP can be 8, 16 or 32, rss is disabled by default.

In Linux* user space, the application can display statistics with the number of packets received on each queue. To have the application display the statistics, send a SIGHUP signal to the running application process.

The VMDq Forwarding sample application is in many ways simpler than the L2 Forwarding application (see *L2 Forwarding Sample Application (in Real and Virtualized Environments)*) as it performs unidirectional L2 forwarding of packets from one port to a second port. No command-line options are taken by this application apart from the standard EAL command-line options.

4.34.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `vmdq` sub-directory.

4.34.3 Running the Application

To run the example in a Linux environment:

```
user@target:~$ ./build/vmdq_app -l 0-3 -n 4 -- -p 0x3 --nb-pools 16
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.34.4 Explanation

The following sections provide some explanation of the code.

Initialization

The EAL, driver and PCI configuration is performed largely as in the L2 Forwarding sample application, as is the creation of the mbuf pool. See *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. Where this example application differs is in the configuration of the NIC port for RX.

The VMDq hardware feature is configured at port initialization time by setting the appropriate values in the rte_eth_conf structure passed to the rte_eth_dev_configure() API. Initially in the application, a default structure is provided for VMDq configuration to be filled in later by the application.

```
/* empty vmdq configuration structure. Filled in programmatically */
static const struct rte_eth_conf vmdq_conf_default = {
    .rxmode = {
        .mq_mode        = ETH_MQ_RX_VMDQ_ONLY,
        .split_hdr_size = 0,
    },

    .txmode = {
        .mq_mode = ETH_MQ_TX_NONE,
    },
    .rx_adv_conf = {
        /*
        * should be overridden separately in code with
        * appropriate values
        */
        .vmdq_rx_conf = {
            .nb_queue_pools = ETH_8_POOLS,
            .enable_default_pool = 0,
            .default_pool = 0,
            .nb_pool_maps = 0,
            .pool_map = {{0, 0},},
        },
    },
};
```

The get_eth_conf() function fills in an rte_eth_conf structure with the appropriate values, based on the global vlan_tags array. For the VLAN IDs, each one can be allocated to possibly multiple pools of queues. For destination MAC, each VMDq pool will be assigned with a MAC address. In this sample, each VMDq pool is assigned to the MAC like 52:54:00:12:<port_id>:<pool_id>, that is, the MAC of VMDq pool 2 on port 1 is 52:54:00:12:01:02.

```
const uint16_t vlan_tags[] = {
    0,  1,  2,  3,  4,  5,  6,  7,
    8,  9, 10, 11,  12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, 23,
    24, 25, 26, 27, 28, 29, 30, 31,
    32, 33, 34, 35, 36, 37, 38, 39,
    40, 41, 42, 43, 44, 45, 46, 47,
    48, 49, 50, 51, 52, 53, 54, 55,
    56, 57, 58, 59, 60, 61, 62, 63,
};

/* pool mac addr template, pool mac addr is like: 52 54 00 12 port# pool# */
static struct rte_ether_addr pool_addr_template = {
    .addr_bytes = {0x52, 0x54, 0x00, 0x12, 0x00, 0x00}
};

/*
 * Builds up the correct configuration for vmdq based on the vlan tags array
 * given above, and determine the queue number and pool map number according to
 * valid pool number
 */
static inline int
get_eth_conf(struct rte_eth_conf *eth_conf, uint32_t num_pools)
{
    struct rte_eth_vmdq_rx_conf conf;
    unsigned i;

    conf.nb_queue_pools = (enum rte_eth_nb_pools)num_pools;
    conf.nb_pool_maps = num_pools;
    conf.enable_default_pool = 0;
    conf.default_pool = 0; /* set explicit value, even if not used */

    for (i = 0; i < conf.nb_pool_maps; i++) {
        conf.pool_map[i].vlan_id = vlan_tags[i];
        conf.pool_map[i].pools = (1UL << (i % num_pools));
    }

    (void)(rte_memcpy(eth_conf, &vmdq_conf_default, sizeof(*eth_conf)));
    (void)(rte_memcpy(&eth_conf->rx_adv_conf.vmdq_rx_conf, &conf,
        sizeof(eth_conf->rx_adv_conf.vmdq_rx_conf)));
    return 0;
}

......

/*
 * Set mac for each pool.
 * There is no default mac for the pools in i40.
 * Removes this after i40e fixes this issue.
 */
for (q = 0; q < num_pools; q++) {
    struct rte_ether_addr mac;
    mac = pool_addr_template;
    mac.addr_bytes[4] = port;
    mac.addr_bytes[5] = q;
    printf("Port %u vmdq pool %u set mac %02x:%02x:%02x:%02x:%02x:%02x\n",
            port, q,
            mac.addr_bytes[0], mac.addr_bytes[1],
            mac.addr_bytes[2], mac.addr_bytes[3],
            mac.addr_bytes[4], mac.addr_bytes[5]);
    retval = rte_eth_dev_mac_addr_add(port, &mac,
                    q + vmdq_pool_base);
```

(continues on next page)

(continued from previous page)

```
    if (retval) {
            printf("mac addr add failed at pool %d\n", q);
            return retval;
    }
}
```

Once the network port has been initialized using the correct VMDq values, the initialization of the port's RX and TX hardware rings is performed similarly to that in the L2 Forwarding sample application. See *L2 Forwarding Sample Application (in Real and Virtualized Environments)* for more information.

Statistics Display

When run in a Linux environment, the VMDq Forwarding sample application can display statistics showing the number of packets read from each RX queue. This is provided by way of a signal handler for the SIGHUP signal, which simply prints to standard output the packet counts in grid form. Each row of the output is a single pool with the columns being the queue number within that pool.

To generate the statistics output, use the following command:

```
user@host$ sudo killall -HUP vmdq_app
```

Please note that the statistics output will appear on the terminal where the vmdq_app is running, rather than the terminal from which the HUP signal was sent.

4.35 Vhost Sample Application

The vhost sample application demonstrates integration of the Data Plane Development Kit (DPDK) with the Linux* KVM hypervisor by implementing the vhost-net offload API. The sample application performs simple packet switching between virtual machines based on Media Access Control (MAC) address or Virtual Local Area Network (VLAN) tag. The splitting of Ethernet traffic from an external switch is performed in hardware by the Virtual Machine Device Queues (VMDQ) and Data Center Bridging (DCB) features of the Intel® 82599 10 Gigabit Ethernet Controller.

4.35.1 Testing steps

This section shows the steps how to test a typical PVP case with this vhost-switch sample, whereas packets are received from the physical NIC port first and enqueued to the VM's Rx queue. Through the guest testpmd's default forwarding mode (io forward), those packets will be put into the Tx queue. The vhost-switch example, in turn, gets the packets and puts back to the same physical NIC port.

Build

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `vhost` sub-directory.

Note: In this example, you need build DPDK both on the host and inside guest.

Start the vswitch example

```
./vhost-switch -l 0-3 -n 4 --socket-mem 1024  \
     -- --socket-file /tmp/sock0 --client \
     ...
```

Check the *Parameters* section for the explanations on what do those parameters mean.

Start the VM

```
qemu-system-x86_64 -machine accel=kvm -cpu host \
    -m $mem -object memory-backend-file,id=mem,size=$mem,mem-path=/dev/hugepages,share=on \
            -mem-prealloc -numa node,memdev=mem \
    \
    -chardev socket,id=char1,path=/tmp/sock0,server \
    -netdev type=vhost-user,id=hostnet1,chardev=char1  \
    -device virtio-net-pci,netdev=hostnet1,id=net1,mac=52:54:00:00:00:14 \
    ...
```

Note: For basic vhost-user support, QEMU 2.2 (or above) is required. For some specific features, a higher version might be need. Such as QEMU 2.7 (or above) for the reconnect feature.

Run testpmd inside guest

Make sure you have DPDK built inside the guest. Also make sure the corresponding virtio-net PCI device is bond to a uio driver, which could be done by:

```
modprobe uio_pci_generic
$RTE_SDK/usertools/dpdk-devbind.py -b uio_pci_generic 0000:00:04.0
```

Then start testpmd for packet forwarding testing.

```
./x86_64-native-gcc/app/testpmd -l 0-1 -- -i
> start tx_first
```

4.35.2 Inject packets

While a virtio-net is connected to vhost-switch, a VLAN tag starts with 1000 is assigned to it. So make sure configure your packet generator with the right MAC and VLAN tag, you should be able to see following log from the vhost-switch console. It means you get it work:

```
VHOST_DATA: (0) mac 52:54:00:00:00:14 and vlan 1000 registered
```

4.35.3 Parameters

–socket-file path Specifies the vhost-user socket file path.

–client DPDK vhost-user will act as the client mode when such option is given. In the client mode, QEMU will create the socket file. Otherwise, DPDK will create it. Put simply, it's the server to create the socket file.

–vm2vm mode The vm2vm parameter sets the mode of packet switching between guests in the host.

- 0 disables vm2vm, implying that VM's packets will always go to the NIC port.
- 1 means a normal mac lookup packet routing.
- 2 means hardware mode packet forwarding between guests, it allows packets go to the NIC port, hardware L2 switch will determine which guest the packet should forward to or need send to external, which bases on the packet destination MAC address and VLAN tag.

–mergeable 0|1 Set 0/1 to disable/enable the mergeable Rx feature. It's disabled by default.

–stats interval The stats parameter controls the printing of virtio-net device statistics. The parameter specifies an interval (in unit of seconds) to print statistics, with an interval of 0 seconds disabling statistics.

–rx-retry 0|1 The rx-retry option enables/disables enqueue retries when the guests Rx queue is full. This feature resolves a packet loss that is observed at high data rates, by allowing it to delay and retry in the receive path. This option is enabled by default.

–rx-retry-num num The rx-retry-num option specifies the number of retries on an Rx burst, it takes effect only when rx retry is enabled. The default value is 4.

–rx-retry-delay msec The rx-retry-delay option specifies the timeout (in micro seconds) between retries on an RX burst, it takes effect only when rx retry is enabled. The default value is 15.

–dequeue-zero-copy Dequeue zero copy will be enabled when this option is given. it is worth to note that if NIC is bound to driver with iommu enabled, dequeue zero copy cannot work at VM2NIC mode (vm2vm=0) due to currently we don't setup iommu dma mapping for guest memory.

–vlan-strip 0|1 VLAN strip option is removed, because different NICs have different behaviors when disabling VLAN strip. Such feature, which heavily depends on hardware, should be removed from this example to reduce confusion. Now, VLAN strip is enabled and cannot be disabled.

–builtin-net-driver A very simple vhost-user net driver which demonstrates how to use the generic vhost APIs will be used when this option is given. It is disabled by default.

4.35.4 Common Issues

- QEMU fails to allocate memory on hugetlbfs, with an error like the following:

```
file_ram_alloc: can't mmap RAM pages: Cannot allocate memory
```

  When running QEMU the above error indicates that it has failed to allocate memory for the Virtual Machine on the hugetlbfs. This is typically due to insufficient hugepages being free to support the allocation request. The number of free hugepages can be checked as follows:

```
cat /sys/kernel/mm/hugepages/hugepages-<pagesize>/nr_hugepages
```

  The command above indicates how many hugepages are free to support QEMU's allocation request.

- Failed to build DPDK in VM

  Make sure "-cpu host" QEMU option is given.

- Device start fails if NIC's max queues > the default number of 128

  mbuf pool size is dependent on the MAX_QUEUES configuration, if NIC's max queue number is larger than 128, device start will fail due to insufficient mbuf.

  Change the default number to make it work as below, just set the number according to the NIC's property.

```
make EXTRA_CFLAGS="-DMAX_QUEUES=320"
```

- Option "builtin-net-driver" is incompatible with QEMU

  QEMU vhost net device start will fail if protocol feature is not negotiated. DPDK virtio-user pmd can be the replacement of QEMU.

- Device start fails when enabling "builtin-net-driver" without memory pre-allocation

  The builtin example doesn't support dynamic memory allocation. When vhost backend enables "builtin-net-driver", "–socket-mem" option should be added at virtio-user pmd side as a startup item.

4.36 Vhost_blk Sample Application

The vhost_blk sample application implemented a simple block device, which used as the backend of Qemu vhost-user-blk device. Users can extend the exist example to use other type of block device(e.g. AIO) besides memory based block device. Similar with vhost-user-net device, the sample application used domain socket to communicate with Qemu, and the virtio ring (split or packed format) was processed by vhost_blk sample application.

The sample application reuse lots codes from SPDK(Storage Performance Development Kit, https://github.com/spdk/spdk) vhost-user-blk target, for DPDK vhost library used in storage area, user can take SPDK as reference as well.

4.36.1 Testing steps

This section shows the steps how to start a VM with the block device as fast data path for critical application.

4.36.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `examples` sub-directory.

You will also need to build DPDK both on the host and inside the guest

Start the vhost_blk example

```
./vhost_blk -m 1024
```

Start the VM

```
qemu-system-x86_64 -machine accel=kvm \
    -m $mem -object memory-backend-file,id=mem,size=$mem,\
    mem-path=/dev/hugepages,share=on -numa node,memdev=mem \
    -drive file=os.img,if=none,id=disk \
    -device ide-hd,drive=disk,bootindex=0 \
    -chardev socket,id=char0,reconnect=1,path=/tmp/vhost.socket \
    -device vhost-user-blk-pci,packed=on,chardev=char0,num-queues=1 \
    ...
```

Note: You must check whether your Qemu can support "vhost-user-blk" or not, Qemu v4.0 or newer version is required. reconnect=1 means live recovery support that qemu can reconnect vhost_blk after we restart vhost_blk example. packed=on means the device support packed ring but need the guest kernel version >= 5.0. Now Qemu commit 9bb73502321d46f4d320fa17aa38201445783fc4 both support the vhost-blk reconnect and packed ring.

4.37 Vhost_Crypto Sample Application

The vhost_crypto sample application implemented a simple Crypto device, which used as the backend of Qemu vhost-user-crypto device. Similar with vhost-user-net and vhost-user-scsi device, the sample application used domain socket to communicate with Qemu, and the virtio ring was processed by vhost_crypto sample application.

4.37.1 Testing steps

This section shows the steps how to start a VM with the crypto device as fast data path for critical application.

4.37.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `examples` sub-directory.

Start the vhost_crypto example

```
./vhost_crypto [EAL options] --
            --config (lcore,cdev-id,queue-id)[,(lcore,cdev-id,queue-id)]
            --socket-file lcore,PATH
            [--zero-copy]
            [--guest-polling]
```

where,

- config (lcore,cdev-id,queue-id): build the lcore-cryptodev id-queue id connection. Once specified, the specified lcore will only work with specified cryptodev's queue.
- socket-file lcore,PATH: the path of UNIX socket file to be created and the lcore id that will deal with the all workloads of the socket. Multiple instances of this config item is supported and one lcore supports processing multiple sockets.
- zero-copy: the presence of this item means the ZERO-COPY feature will be enabled. Otherwise it is disabled. PLEASE NOTE the ZERO-COPY feature is still in experimental stage and may cause the problem like segmentation fault. If the user wants to use LKCF in the guest, this feature shall be turned off.
- guest-polling: the presence of this item means the application assumes the guest works in polling mode, thus will NOT notify the guest completion of processing.

The application requires that crypto devices capable of performing the specified crypto operation are available on application initialization. This means that HW crypto device/s must be bound to a DPDK driver or a SW crypto device/s (virtual crypto PMD) must be created (using –vdev).

Start the VM

```
qemu-system-x86_64 -machine accel=kvm \
    -m $mem -object memory-backend-file,id=mem,size=$mem,\
    mem-path=/dev/hugepages,share=on -numa node,memdev=mem \
    -drive file=os.img,if=none,id=disk \
    -device ide-hd,drive=disk,bootindex=0 \
    -chardev socket,id={chardev_id},path={PATH} \
    -object cryptodev-vhost-user,id={obj_id},chardev={chardev_id} \
    -device virtio-crypto-pci,id={dev_id},cryptodev={obj_id} \
    ...
```

Note: You must check whether your Qemu can support "vhost-user-crypto" or not.

4.38 Vdpa Sample Application

The vdpa sample application creates vhost-user sockets by using the vDPA backend. vDPA stands for vhost Data Path Acceleration which utilizes virtio ring compatible devices to serve virtio driver directly to enable datapath acceleration. As vDPA driver can help to set up vhost datapath, this application doesn't need to launch dedicated worker threads for vhost enqueue/dequeue operations.

4.38.1 Testing steps

This section shows the steps of how to start VMs with vDPA vhost-user backend and verify network connection & live migration.

Build

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `vdpa` sub-directory.

Start the vdpa example

```
./vdpa [EAL options]  -- [--client] [--interactive|-i] or [--iface SOCKET_PATH]
```

where

- –client means running vdpa app in client mode, in the client mode, QEMU needs to run as the server mode and take charge of socket file creation.
- –iface specifies the path prefix of the UNIX domain socket file, e.g. /tmp/vhost-user-, then the socket files will be named as /tmp/vhost-user-<n> (n starts from 0).
- –interactive means run the vdpa sample in interactive mode, currently 4 internal cmds are supported:
  1. help: show help message
  2. list: list all available vdpa devices
  3. create: create a new vdpa port with socket file and vdpa device address
  4. quit: unregister vhost driver and exit the application

Take IFCVF driver for example:

```
./vdpa -c 0x2 -n 4 --socket-mem 1024,1024 \
        -w 0000:06:00.3,vdpa=1 -w 0000:06:00.4,vdpa=1 \
        -- --interactive
```

Note: Here 0000:06:00.3 and 0000:06:00.4 refer to virtio ring compatible devices, and we need to bind vfio-pci to them before running vdpa sample.

- modprobe vfio-pci
- ./usertools/dpdk-devbind.py -b vfio-pci 06:00.3 06:00.4

Then we can create 2 vdpa ports in interactive cmdline.

```
vdpa> list
device id       device address  queue num       supported features
0               0000:06:00.3    1               0x14c238020
1               0000:06:00.4    1               0x14c238020
2               0000:06:00.5    1               0x14c238020

vdpa> create /tmp/vdpa-socket0 0000:06:00.3
vdpa> create /tmp/vdpa-socket1 0000:06:00.4
```

Start the VMs

```
qemu-system-x86_64 -cpu host -enable-kvm \
<snip>
-mem-prealloc \
-chardev socket,id=char0,path=<socket_file created in above steps> \
-netdev type=vhost-user,id=vdpa,chardev=char0 \
-device virtio-net-pci,netdev=vdpa,mac=00:aa:bb:cc:dd:ee,page-per-vq=on \
```

After the VMs launches, we can login the VMs and configure the ip, verify the network connection via ping or netperf.

Note: Suggest to use QEMU 3.0.0 which extends vhost-user for vDPA.

Live Migration

vDPA supports cross-backend live migration, user can migrate SW vhost backend VM to vDPA backend VM and vice versa. Here are the detailed steps. Assume A is the source host with SW vhost VM and B is the destination host with vDPA.

1. Start vdpa sample and launch a VM with exact same parameters as the VM on A, in migration-listen mode:

```
B: <qemu-command-line> -incoming tcp:0:4444 (or other PORT))
```

2. Start the migration (on source host):

```
A: (qemu) migrate -d tcp:<B ip>:4444 (or other PORT)
```

3. Check the status (on source host):

```
A: (qemu) info migrate
```

4.39 Internet Protocol (IP) Pipeline Application

4.39.1 Application overview

The *Internet Protocol (IP) Pipeline* application is intended to be a vehicle for rapid development of packet processing applications on multi-core CPUs.

Following OpenFlow and P4 design principles, the application can be used to create functional blocks called pipelines out of input/output ports, tables and actions in a modular way. Multiple pipelines can be inter-connected through packet queues to create complete applications (super-pipelines).

The pipelines are mapped to application threads, with each pipeline executed by a single thread and each thread able to run one or several pipelines. The possibilities of creating pipelines out of ports, tables and actions, connecting multiple pipelines together and mapping the pipelines to execution threads are endless, therefore this application can be seen as a true application generator.

Pipelines are created and managed through Command Line Interface (CLI):

- Any standard TCP client (e.g. telnet, netcat, custom script, etc) is typically able to connect to the application, send commands through the network and wait for the response before pushing the next command.
- **All the application objects are created and managed through CLI commands:**
  - 'Primitive' objects used to create pipeline ports: memory pools, links (i.e. network interfaces), SW queues, traffic managers, etc.
  - Action profiles: used to define the actions to be executed by pipeline input/output ports and tables.
  - Pipeline components: input/output ports, tables, pipelines, mapping of pipelines to execution threads.

4.39.2 Running the application

The application startup command line is:

```
ip_pipeline [EAL_ARGS] -- [-s SCRIPT_FILE] [-h HOST] [-p PORT]
```

The application startup arguments are:

`-s SCRIPT_FILE`

- Optional: Yes
- Default: Not present
- Argument: Path to the CLI script file to be run at application startup. No CLI script file will run at startup if this argument is not present.

`-h HOST`

- Optional: Yes
- Default: `0.0.0.0`
- Argument: IP Address of the host running ip pipeline application to be used by remote TCP based client (telnet, netcat, etc.) for connection.

`-p PORT`

- Optional: Yes
- Default: `8086`
- Argument: TCP port number at which the ip pipeline is running. This port number should be used by remote TCP client (such as telnet, netcat, etc.) to connect to host application.

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The following is an example command to run ip pipeline application configured for layer 2 forwarding:

```
$ ./build/ip_pipeline -c 0x3 -- -s examples/route_ecmp.cli
```

The application should start successfully and display as follows:

```
EAL: Detected 40 lcore(s)
EAL: Detected 2 NUMA nodes
EAL: Multi-process socket /var/run/.rte_unix
EAL: Probing VFIO support...
EAL: PCI device 0000:02:00.0 on NUMA socket 0
EAL:   probe driver: 8086:10fb net_ixgbe
...
```

To run remote client (e.g. telnet) to communicate with the ip pipeline application:

```
$ telnet 127.0.0.1 8086
```

When running a telnet client as above, command prompt is displayed:

```
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

Welcome to IP Pipeline!

pipeline>
```

Once application and telnet client start running, messages can be sent from client to application. At any stage, telnet client can be terminated using the quit command.

4.39.3 Application stages

Initialization

During this stage, EAL layer is initialised and application specific arguments are parsed. Furthermore, the data structures (i.e. linked lists) for application objects are initialized. In case of any initialization error, an error message is displayed and the application is terminated.

Run-time

The master thread is creating and managing all the application objects based on CLI input.

Each data plane thread runs one or several pipelines previously assigned to it in round-robin order. Each data plane thread executes two tasks in time-sharing mode:

1. *Packet processing task*: Process bursts of input packets read from the pipeline input ports.
2. *Message handling task*: Periodically, the data plane thread pauses the packet processing task and polls for request messages send by the master thread. Examples: add/remove pipeline to/from current data plane thread, add/delete rules to/from given table of a specific pipeline owned by the current data plane thread, read statistics, etc.

4.39.4 Examples

Table 4.3: Pipeline examples provided with the application

Name	Table(s)	Actions	Messages
L2fwd Note: Implemented using pipeline with a simple pass-through connection between input and output ports.	Stub	Forward	1. Mempool create 2. Link create 3. Pipeline create 4. Pipeline port in/out 5. Pipeline table 6. Pipeline port in table 7. Pipeline enable 8. Pipeline table rule add
Flow classification	Exact match • **Key = byte array** (16 bytes) • Offset = 278 • Table size = 64K	Forward	1. Mempool create 2. Link create 3. Pipeline create 4. Pipeline port in/out 5. Pipeline table 6. Pipeline port in table 7. Pipeline enable 8. Pipeline table rule add default 9. Pipeline table rule add
KNI	Stub	Forward	1. Mempool create 2. Link create 3. Pipeline create 4. Pipeline port in/out 5. Pipeline table 6. Pipeline port in table 7. Pipeline enable 8. Pipeline table rule add
Firewall	ACL • Key = n-tuple • Offset = 270 • Table size = 4K	Allow/Drop	1. Mempool create 2. Link create 3. Pipeline create 4. Pipeline port in/out 5. Pipeline table 6. Pipeline port in table 7. Pipeline enable 8. Pipeline table rule add default 9. Pipeline table rule add
IP routing	LPM (IPv4)	Forward	1. Mempool Create

4.39.5 Command Line Interface (CLI)

Link

Link configuration

```
link <link_name>
 dev <device_name>|port <port_id>
 rxq <n_queues> <queue_size> <mempool_name>
 txq <n_queues> <queue_size> promiscuous on | off
 [rss <qid_0> ... <qid_n>]
```

Note: The PCI device name must be specified in the Domain:Bus:Device.Function format.

Mempool

Mempool create

```
mempool <mempool_name> buffer <buffer_size>
pool <pool_size> cache <cache_size> cpu <cpu_id>
```

Software queue

Create software queue

```
swq <swq_name> size <size> cpu <cpu_id>
```

Traffic manager

Add traffic manager subport profile

```
tmgr subport profile
 <tb_rate> <tb_size>
 <tc0_rate> <tc1_rate> <tc2_rate> <tc3_rate> <tc4_rate>
 <tc5_rate> <tc6_rate> <tc7_rate> <tc8_rate>
 <tc9_rate> <tc10_rate> <tc11_rate> <tc12_rate>
 <tc_period>
 pps <n_pipes_per_subport>
 qsize <qsize_tc0> <qsize_tc1> <qsize_tc2>
 <qsize_tc3> <qsize_tc4> <qsize_tc5> <qsize_tc6>
 <qsize_tc7> <qsize_tc8> <qsize_tc9> <qsize_tc10>
 <qsize_tc11> <qsize_tc12>
```

Add traffic manager pipe profile

```
tmgr pipe profile
 <tb_rate> <tb_size>
 <tc0_rate> <tc1_rate> <tc2_rate> <tc3_rate> <tc4_rate>
 <tc5_rate> <tc6_rate> <tc7_rate> <tc8_rate>
 <tc9_rate> <tc10_rate> <tc11_rate> <tc12_rate>
 <tc_period>
 <tc_ov_weight>
 <wrr_weight0..3>
```

Create traffic manager port

```
tmgr <tmgr_name>
 rate <rate>
 spp <n_subports_per_port>
 fo <frame_overhead>
 mtu <mtu>
 cpu <cpu_id>
```

Configure traffic manager subport

```
tmgr <tmgr_name>
 subport <subport_id>
 profile <subport_profile_id>
```

Configure traffic manager pipe

```
tmgr <tmgr_name>
 subport <subport_id>
 pipe from <pipe_id_first> to <pipe_id_last>
 profile <pipe_profile_id>
```

Tap

Create tap port

```
tap <name>
```

Kni

Create kni port

```
kni <kni_name>
 link <link_name>
 mempool <mempool_name>
 [thread <thread_id>]
```

Cryptodev

Create cryptodev port

```
cryptodev <cryptodev_name>
 dev <DPDK Cryptodev PMD name>
 queue <n_queues> <queue_size>
```

Action profile

Create action profile for pipeline input port

```
port in action profile <profile_name>
 [filter match | mismatch offset <key_offset> mask <key_mask> key <key_value> port
↪<port_id>]
 [balance offset <key_offset> mask <key_mask> port <port_id0> ... <port_id15>]
```

Create action profile for the pipeline table

```
table action profile <profile_name>
 ipv4 | ipv6
 offset <ip_offset>
 fwd
 [balance offset <key_offset> mask <key_mask> outoffset <out_offset>]
 [meter srtcm | trtcm
     tc <n_tc>
     stats none | pkts | bytes | both]
 [tm spp <n_subports_per_port> pps <n_pipes_per_subport>]
 [encap ether | vlan | qinq | mpls | pppoe]
 [nat src | dst
     proto udp | tcp]
 [ttl drop | fwd
     stats none | pkts]
 [stats pkts | bytes | both]
 [sym_crypto cryptodev <cryptodev_name>
     mempool_create <mempool_name> mempool_init <mempool_name>]
 [time]
```

Pipeline

Create pipeline

```
pipeline <pipeline_name>
 period <timer_period_ms>
 offset_port_id <offset_port_id>
 cpu <cpu_id>
```

Create pipeline input port

```
pipeline <pipeline_name> port in
 bsz <burst_size>
 link <link_name> rxq <queue_id>
 | swq <swq_name>
 | tmgr <tmgr_name>
 | tap <tap_name> mempool <mempool_name> mtu <mtu>
 | kni <kni_name>
 | source mempool <mempool_name> file <file_name> bpp <n_bytes_per_pkt>
 [action <port_in_action_profile_name>]
 [disabled]
```

Create pipeline output port

```
pipeline <pipeline_name> port out
 bsz <burst_size>
 link <link_name> txq <txq_id>
 | swq <swq_name>
```

(continues on next page)

(continued from previous page)

```
 | tmgr <tmgr_name>
 | tap <tap_name>
 | kni <kni_name>
 | sink [file <file_name> pkts <max_n_pkts>]
```

Create pipeline table

```
pipeline <pipeline_name> table
     match
     acl
         ipv4 | ipv6
         offset <ip_header_offset>
         size <n_rules>
     | array
         offset <key_offset>
         size <n_keys>
     | hash
         ext | lru
         key <key_size>
         mask <key_mask>
         offset <key_offset>
         buckets <n_buckets>
         size <n_keys>
     | lpm
         ipv4 | ipv6
         offset <ip_header_offset>
         size <n_rules>
     | stub
 [action <table_action_profile_name>]
```

Connect pipeline input port to table

```
pipeline <pipeline_name> port in <port_id> table <table_id>
```

Display statistics for specific pipeline input port, output port or table

```
pipeline <pipeline_name> port in <port_id> stats read [clear]
pipeline <pipeline_name> port out <port_id> stats read [clear]
pipeline <pipeline_name> table <table_id> stats read [clear]
```

Enable given input port for specific pipeline instance

```
pipeline <pipeline_name> port out <port_id> disable
```

Disable given input port for specific pipeline instance

```
pipeline <pipeline_name> port out <port_id> disable
```

Add default rule to table for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> rule add
   match
      default
   action
      fwd
         drop
         | port <port_id>
         | meta
         | table <table_id>
```

Add rule to table for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> rule add

match
   acl
      priority <priority>
      ipv4 | ipv6 <sa> <sa_depth> <da> <da_depth>
      <sp0> <sp1> <dp0> <dp1> <proto>
   | array <pos>
   | hash
      raw <key>
      | ipv4_5tuple <sa> <da> <sp> <dp> <proto>
      | ipv6_5tuple <sa> <da> <sp> <dp> <proto>
      | ipv4_addr <addr>
      | ipv6_addr <addr>
      | qinq <svlan> <cvlan>
   | lpm
      ipv4 | ipv6 <addr> <depth>

action
   fwd
      drop
      | port <port_id>
      | meta
      | table <table_id>
   [balance <out0> ... <out7>]
   [meter
      tc0 meter <meter_profile_id> policer g <pa> y <pa> r <pa>
      [tc1 meter <meter_profile_id> policer g <pa> y <pa> r <pa>
      tc2 meter <meter_profile_id> policer g <pa> y <pa> r <pa>
      tc3 meter <meter_profile_id> policer g <pa> y <pa> r <pa>]]
   [tm subport <subport_id> pipe <pipe_id>]
   [encap
      ether <da> <sa>
      | vlan <da> <sa> <pcp> <dei> <vid>
      | qinq <da> <sa> <pcp> <dei> <vid> <pcp> <dei> <vid>
      | mpls unicast | multicast
         <da> <sa>
         label0 <label> <tc> <ttl>
         [label1 <label> <tc> <ttl>
         [label2 <label> <tc> <ttl>
         [label3 <label> <tc> <ttl>]]]
      | pppoe <da> <sa> <session_id>]
   [nat ipv4 | ipv6 <addr> <port>]
   [ttl dec | keep]
   [stats]
   [time]
   [sym_crypto
      encrypt | decrypt
      type
      | cipher
         cipher_algo <algo> cipher_key <key> cipher_iv <iv>
      | cipher_auth
         cipher_algo <algo> cipher_key <key> cipher_iv <iv>
         auth_algo <algo> auth_key <key> digest_size <size>
      | aead
         aead_algo <algo> aead_key <key> aead_iv <iv> aead_aad <aad>
         digest_size <size>
      data_offset <data_offset>]

where:
   <pa> ::= g | y | r | drop
```

Add bulk rules to table for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> rule add bulk <file_name> <n_rules>

Where:
- file_name = path to file
- File line format = match <match> action <action>
```

Delete table rule for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> rule delete
   match <match>
```

Delete default table rule for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> rule delete
   match
      default
```

Add meter profile to the table for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> meter profile <meter_profile_id>
 add srtcm cir <cir> cbs <cbs> ebs <ebs>
 | trtcm cir <cir> pir <pir> cbs <cbs> pbs <pbs>
```

Delete meter profile from the table for specific pipeline instance

```
pipeline <pipeline_name> table <table_id>
 meter profile <meter_profile_id> delete
```

Update the dscp table for meter or traffic manager action for specific pipeline instance

```
pipeline <pipeline_name> table <table_id> dscp <file_name>

Where:
   - file_name = path to file
   - exactly 64 lines
   - File line format = <tc_id> <tc_queue_id> <color>, with <color> as: g | y | r
```

Pipeline enable/disable

Enable given pipeline instance for specific data plane thread

```
thread <thread_id> pipeline <pipeline_name> enable
```

Disable given pipeline instance for specific data plane thread

```
thread <thread_id> pipeline <pipeline_name> disable
```

4.40 Test Pipeline Application

The Test Pipeline application illustrates the use of the DPDK Packet Framework tool suite. Its purpose is to demonstrate the performance of single-table DPDK pipelines.

4.40.1 Overview

The application uses three CPU cores:

- Core A ("RX core") receives traffic from the NIC ports and feeds core B with traffic through SW queues.
- Core B ("Pipeline core") implements a single-table DPDK pipeline whose type is selectable through specific command line parameter. Core B receives traffic from core A through software queues, processes it according to the actions configured in the table entries that are hit by the input packets and feeds it to core C through another set of software queues.
- Core C ("TX core") receives traffic from core B through software queues and sends it to the NIC ports for transmission.

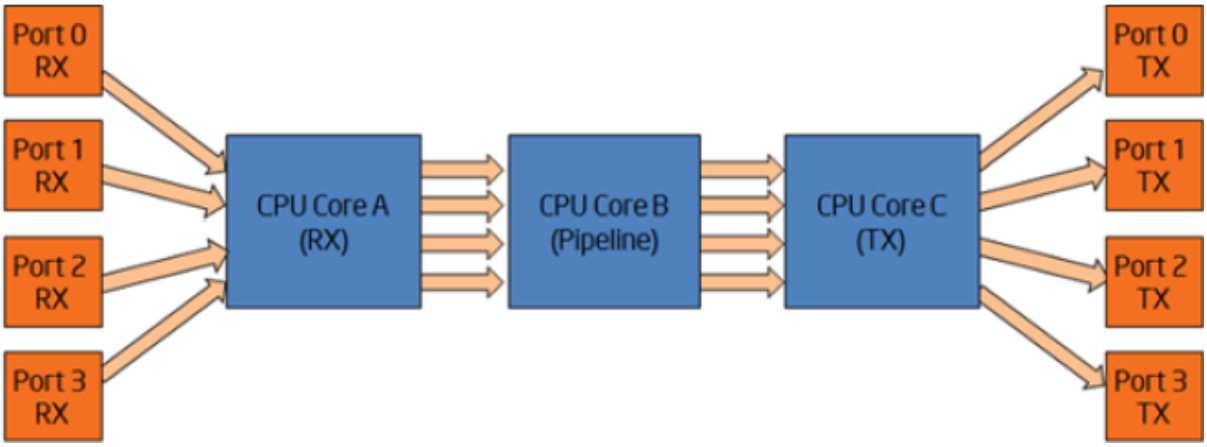


Fig. 4.16: Test Pipeline Application

4.40.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*

The application is located in the `$RTE_SDK/app/test-pipeline` directory.

4.40.3 Running the Application

Application Command Line

The application execution command line is:

```
./test-pipeline [EAL options] -- -p PORTMASK --TABLE_TYPE
```

The -c or -l EAL CPU coremask/corelist option has to contain exactly 3 CPU cores. The first CPU core in the core mask is assigned for core A, the second for core B and the third for core C.

The PORTMASK parameter must contain 2 or 4 ports.

Table Types and Behavior

Table 4.4 describes the table types used and how they are populated.

The hash tables are pre-populated with 16 million keys. For hash tables, the following parameters can be selected:

- **Configurable key size implementation or fixed (specialized) key size implementation (e.g. hash-8-ext or hash-spec-8-ext).** The key size specialized implementations are expected to provide better performance for 8-byte and 16-byte key sizes, while the key-size-non-specialized implementation is expected to provide better performance for larger key sizes;
- **Key size (e.g. hash-spec-8-ext or hash-spec-16-ext).** The available options are 8, 16 and 32 bytes;
- **Table type (e.g. hash-spec-16-ext or hash-spec-16-lru).** The available options are ext (extendable bucket) or lru (least recently used).

Table 4.4: Table Types

#	TABLE_TYPE	Description of Core B Table	Pre-added Table Entries
1	none	Core B is not implementing a DPDK pipeline. Core B is implementing a pass-through from its input set of software queues to its output set of software queues.	N/A
2	stub	Stub table. Core B is implementing the same pass-through functionality as described for the "none" option by using the DPDK Packet Framework by using one stub table for each input NIC port.	N/A
3	hash-[spec]-8-lru	LRU hash table with 8-byte key size and 16 million entries.	16 million entries are successfully added to the hash table with the following key format: [4-byte index, 4 bytes of 0] The action configured for all table entries is "Sendto output port", with the output port index uniformly distributed for the range of output ports. The default table rule (used in the case of a lookup miss) is to drop the packet. At run time, core A is creating the following lookup key and storing it into the packet meta data for core B to use for table lookup: [destination IPv4 address, 4 bytes of 0]
4	hash-[spec]-8-ext	Extendable bucket hash table with 8-byte key size and 16 million entries.	Same as hash-[spec]-8-lru table entries, above.
5	hash-[spec]-16-lru	LRU hash table with 16-byte key size and 16 million entries.	16 million entries are successfully added to the hash table with the following key format: [4-byte index, 12 bytes of 0] The action configured for all table entries is "Send to output port", with the output port index uniformly distributed for the range of output ports. The default table rule (used in the case of a lookup miss) is to drop the packet. At run time, core A is creating the following lookup key and storing it into the packet meta data for core B to use for table lookup: [destination IPv4 address, 12 bytes of 0]
6	hash-[spec]-16-ext	Extendable bucket hash table with 16-byte key size and 16 million entries.	Same as hash-[spec]-16-lru table entries, above.
7	hash-[spec]-32	LRU hash table with 32-byte key size and 16 million entries.	16 million entries are successfully added to the hash table with the following key format: [4-byte index, 28 bytes of 0]

Input Traffic

Regardless of the table type used for the core B pipeline, the same input traffic can be used to hit all table entries with uniform distribution, which results in uniform distribution of packets sent out on the set of output NIC ports. The profile for input traffic is TCP/IPv4 packets with:

- destination IP address as A.B.C.D with A fixed to 0 and B, C,D random
- source IP address fixed to 0.0.0.0
- destination TCP port fixed to 0
- source TCP port fixed to 0

4.41 Eventdev Pipeline Sample Application

The eventdev pipeline sample application is a sample app that demonstrates the usage of the eventdev API using the software PMD. It shows how an application can configure a pipeline and assign a set of worker cores to perform the processing required.

The application has a range of command line arguments allowing it to be configured for various numbers worker cores, stages,queue depths and cycles per stage of work. This is useful for performance testing as well as quickly testing a particular pipeline configuration.

4.41.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `examples` sub-directory.

4.41.2 Running the Application

The application has a lot of command line options. This allows specification of the eventdev PMD to use, and a number of attributes of the processing pipeline options.

An example eventdev pipeline running with the software eventdev PMD using these settings is shown below:

- `-r1`: core mask 0x1 for RX
- `-t1`: core mask 0x1 for TX
- `-e4`: core mask 0x4 for the software scheduler
- `-w FF00`: core mask for worker cores, 8 cores from 8th to 16th
- `-s4`: 4 atomic stages
- `-n0`: process infinite packets (run forever)
- `-c32`: worker dequeue depth of 32
- `-W1000`: do 1000 cycles of work per packet in each stage
- `-D`: dump statistics on exit

```
./build/eventdev_pipeline --vdev event_sw0 -- -r1 -t1 -e4 -w FF00 -s4 -n0 -c32 -W1000 -D
```

The application has some sanity checking built-in, so if there is a function (e.g.; the RX core) which doesn't have a cpu core mask assigned, the application will print an error message:

```
Core part of pipeline was not assigned any cores. This will stall the
pipeline, please check core masks (use -h for details on setting core masks):
        rx: 0
        tx: 1
```

Configuration of the eventdev is covered in detail in the programmers guide, see the Event Device Library section.

4.41.3 Observing the Application

At runtime the eventdev pipeline application prints out a summary of the configuration, and some runtime statistics like packets per second. On exit the worker statistics are printed, along with a full dump of the PMD statistics if required. The following sections show sample output for each of the output types.

Configuration

This provides an overview of the pipeline, scheduling type at each stage, and parameters to options such as how many flows to use and what eventdev PMD is in use. See the following sample output for details:

```
Config:
      ports: 2
      workers: 8
      packets: 0
      priorities: 1
      Queue-prio: 0
      qid0 type: atomic
      Cores available: 44
      Cores used: 10
      Eventdev 0: event_sw
Stages:
      Stage 0, Type Atomic    Priority = 128
      Stage 1, Type Atomic    Priority = 128
      Stage 2, Type Atomic    Priority = 128
      Stage 3, Type Atomic    Priority = 128
```

Runtime

At runtime, the statistics of the consumer are printed, stating the number of packets received, runtime in milliseconds, average mpps, and current mpps.

```
# consumer RX= xxxxxxx, time yyyy ms, avg z.zzz mpps [current w.www mpps]
```

Shutdown

At shutdown, the application prints the number of packets received and transmitted, and an overview of the distribution of work across worker cores.

```
Signal 2 received, preparing to exit...
  worker 12 thread done. RX=4966581 TX=4966581
  worker 13 thread done. RX=4963329 TX=4963329
  worker 14 thread done. RX=4953614 TX=4953614
  worker 0 thread done. RX=0 TX=0
  worker 11 thread done. RX=4970549 TX=4970549
  worker 10 thread done. RX=4986391 TX=4986391
  worker 9 thread done. RX=4970528 TX=4970528
  worker 15 thread done. RX=4974087 TX=4974087
  worker 8 thread done. RX=4979908 TX=4979908
  worker 2 thread done. RX=0 TX=0

Port Workload distribution:
worker 0 :      12.5 % (4979876 pkts)
worker 1 :      12.5 % (4970497 pkts)
worker 2 :      12.5 % (4986359 pkts)
worker 3 :      12.5 % (4970517 pkts)
worker 4 :      12.5 % (4966566 pkts)
worker 5 :      12.5 % (4963297 pkts)
worker 6 :      12.5 % (4953598 pkts)
worker 7 :      12.5 % (4974055 pkts)
```

To get a full dump of the state of the eventdev PMD, pass the `-D` flag to this application. When the app is terminated using `Ctrl+C`, the `rte_event_dev_dump()` function is called, resulting in a dump of the statistics that the PMD provides. The statistics provided depend on the PMD used, see the Event Device Drivers section for a list of eventdev PMDs.

4.42 Distributor Sample Application

The distributor sample application is a simple example of packet distribution to cores using the Data Plane Development Kit (DPDK). It also makes use of Intel Speed Select Technology - Base Frequency (Intel SST-BF) to pin the distributor to the higher frequency core if available.

4.42.1 Overview

The distributor application performs the distribution of packets that are received on an RX_PORT to different cores. When processed by the cores, the destination port of a packet is the port from the enabled port mask adjacent to the one on which the packet was received, that is, if the first four ports are enabled (port mask 0xf), ports 0 and 1 RX/TX into each other, and ports 2 and 3 RX/TX into each other.

This application can be used to benchmark performance using the traffic generator as shown in the figure below.

Fig. 4.17: Performance Benchmarking Setup (Basic Environment)

4.42.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `distributor` sub-directory.

4.42.3 Running the Application

1. The application has a number of command line options:

```
./build/distributor_app [EAL options] -- -p PORTMASK
```

   where,

   - -p PORTMASK: Hexadecimal bitmask of ports to configure

2. To run the application in linux environment with 10 lcores, 4 ports, issue the command:

```
$ ./build/distributor_app -l 1-9,22 -n 4 -- -p f
```

3. Refer to the DPDK Getting Started Guide for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.42.4 Explanation

The distributor application consists of four types of threads: a receive thread (`lcore_rx()`), a distributor thread (`lcore_dist()`), a set of worker threads (`lcore_worker()`), and a transmit thread(`lcore_tx()`). How these threads work together is shown in Fig. 4.18 below. The `main()` function launches threads of these four types. Each thread has a while loop which will be doing processing and which is terminated only upon SIGINT or ctrl+C.

The receive thread receives the packets using `rte_eth_rx_burst()` and will enqueue them to an rte_ring. The distributor thread will dequeue the packets from the ring and assign them to workers (using `rte_distributor_process()` API). This assignment is based on the tag (or flow ID) of the packet - indicated by the hash field in the mbuf. For IP traffic, this field is automatically filled by the NIC with the "usr" hash value for the packet, which works as a per-flow tag. The distributor thread communicates with the worker threads using a cache-line swapping mechanism, passing up to 8 mbuf pointers at a time (one cache line) to each worker.

More than one worker thread can exist as part of the application, and these worker threads do simple packet processing by requesting packets from the distributor, doing a simple XOR operation on the input port mbuf field (to indicate the output port which will be used later for packet transmission) and then finally returning the packets back to the distributor thread.

The distributor thread will then call the distributor api `rte_distributor_returned_pkts()` to get the processed packets, and will enqueue them to another rte_ring for transfer to the TX thread for transmission on the output port. The transmit thread will dequeue the packets from the ring and transmit them on the output port specified in packet mbuf.

Users who wish to terminate the running of the application have to press ctrl+C (or send SIGINT to the app). Upon this signal, a signal handler provided in the application will terminate all running threads gracefully and print final statistics to the user.

Fig. 4.18: Distributor Sample Application Layout

4.42.5 Intel SST-BF Support

In DPDK 19.05, support was added to the power management library for Intel-SST-BF, a technology that allows some cores to run at a higher frequency than others. An application note for Intel SST-BF is available, and is entitled Intel Speed Select Technology – Base Frequency - Enhancing Performance

The distributor application was also enhanced to be aware of these higher frequency SST-BF cores, and when starting the application, if high frequency SST-BF cores are present in the core mask, the application will identify these cores and pin the workloads appropriately. The distributor core is usually the bottleneck, so this is given first choice of the high frequency SST-BF cores, followed by the rx core and the tx core.

4.42.6 Debug Logging Support

Debug logging is provided as part of the application; the user needs to uncomment the line "#define DEBUG" defined in start of the application in main.c to enable debug logs.

4.42.7 Statistics

The main function will print statistics on the console every second. These statistics include the number of packets enqueued and dequeued at each stage in the application, and also key statistics per worker, including how many packets of each burst size (1-8) were sent to each worker thread.

4.42.8 Application Initialization

Command line parsing is done in the same way as it is done in the L2 Forwarding Sample Application. See *Command Line Arguments*.

Mbuf pool initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See *Mbuf Pool Initialization*.

Driver Initialization is done in same way as it is done in the L2 Forwarding Sample Application. See *Driver Initialization*.

RX queue initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See *RX Queue Initialization*.

TX queue initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See *TX Queue Initialization*.

4.43 Virtual Machine Power Management Application

Applications running in virtual environments have an abstract view of the underlying hardware on the host. Specifically, applications cannot see the binding of virtual components to physical hardware. When looking at CPU resourcing, the pinning of Virtual CPUs (vCPUs) to Physical CPUs (pCPUs) on the host is not apparent to an application and this pinning may change over time. In addition, operating systems on Virtual Machines (VMs) do not have the ability to govern their own power policy. The Machine Specific Registers (MSRs) for enabling P-state transitions are not exposed to the operating systems running on the VMs.

The solution demonstrated in this sample application shows an example of how a DPDK application can indicate its processing requirements using VM-local only information (vCPU/lcore, and so on) to a host resident VM Power Manager. The VM Power Manager is responsible for:

- **Accepting requests for frequency changes for a vCPU**
- **Translating the vCPU to a pCPU using libvirt**
- **Performing the change in frequency**

This application demonstrates the following features:

- **The handling of VM application requests to change frequency.** VM applications can request frequency changes for a vCPU. The VM Power Management Application uses libvirt to translate that virtual CPU (vCPU) request to a physical CPU (pCPU) request and performs the frequency change.
- **The acceptance of power management policies from VM applications.** A VM application can send a policy to the host application. The policy contains rules that define the power management behaviour of the VM. The host application then applies the rules of the policy independent of the VM application. For example, the policy can contain time-of-day information for busy/quiet periods, and the host application can scale up/down the relevant cores when required. See *Command Line Options Available When Sending a Policy to the Host* for information on setting policy values.
- **Out-of-band monitoring of workloads using core hardware event counters.** The host application can manage power for an application by looking at the event counters of the cores and taking action based on the branch miss/hit ratio. See *Command Line Options for Enabling Out-of-band Branch Ratio Monitoring*.

  Note: This functionality also applies in non-virtualised environments.

In addition to the `librte_power` library used on the host, the application uses a special version of `librte_power` on each VM, which directs frequency changes and policies to the host monitor rather than the APCI `cpufreq` `sysfs` interface used on the host in non-virtualised environments.

Fig. 4.19: Highlevel Solution

In the above diagram, the DPDK Applications are shown running in virtual machines, and the VM Power Monitor application is shown running in the host.

DPDK VM Application

- Reuse `librte_power` interface, but uses an implementation that forwards frequency requests to the host using a `virtio-serial` channel
- Each lcore has exclusive access to a single channel

- Sample application reuses `l3fwd_power`
- A CLI for changing frequency from within a VM is also included

VM Power Monitor

- Accepts VM commands over `virtio-serial` endpoints, monitored using `epoll`
- Commands include the virtual core to be modified, using `libvirt` to get the physical core mapping
- Uses `librte_power` to affect frequency changes using Linux userspace power governor (`acpi_cpufreq` OR `intel_pstate` driver)
- CLI: For adding VM channels to monitor, inspecting and changing channel state, manually altering CPU frequency. Also allows for the changings of vCPU to pCPU pinning

4.43.1 Sample Application Architecture Overview

The VM power management solution employs `qemu-kvm` to provide communications channels between the host and VMs in the form of a `virtio-serial` connection that appears as a para-virtualised serial device on a VM and can be configured to use various backends on the host. For this example, the configuration of each `virtio-serial` endpoint on the host as an `AF_UNIX` file socket, supporting poll/select and `epoll` for event notification. In this example, each channel endpoint on the host is monitored for `EPOLLIN` events using `epoll`. Each channel is specified as `qemu-kvm` arguments or as `libvirt` XML for each VM, where each VM can have several channels up to a maximum of 64 per VM. In this example, each DPDK lcore on a VM has exclusive access to a channel.

To enable frequency changes from within a VM, the VM forwards a `librte_power` request over the `virtio-serial` channel to the host. Each request contains the vCPU and power command (scale up/down/min/max). The API for the host `librte_power` and guest `librte_power` is consistent across environments, with the selection of VM or host implementation determined automatically at runtime based on the environment. On receiving a request, the host translates the vCPU to a pCPU using the libvirt API before forwarding it to the host `librte_power`.

In addition to the ability to send power management requests to the host, a VM can send a power management policy to the host. In some cases, using a power management policy is a preferred option because it can eliminate possible latency issues that can occur when sending power management requests. Once the VM sends the policy to the host, the VM no longer needs to worry about power management, because the host now manages the power for the VM based on the policy. The policy can specify power behavior that is based on incoming traffic rates or time-of-day power adjustment (busy/quiet hour power adjustment for example). See *Command Line Options Available When Sending a Policy to the Host* for more information.

One method of power management is to sense how busy a core is when processing packets and adjusting power accordingly. One technique for doing this is to monitor the ratio of the branch miss to branch hits counters and scale the core power accordingly. This technique is based on the premise that when a core is not processing packets, the ratio of branch misses to branch hits is very low, but when the core is processing packets, it is measurably higher. The implementation of this capability is as a policy of type `BRANCH_RATIO`. See *Command Line Options Available When Sending a Policy to the Host* for more information on using the BRANCH_RATIO policy option.

A JSON interface enables the specification of power management requests and policies in JSON format. The JSON interfaces provide a more convenient and more easily interpreted interface for the specification of requests and policies. See *JSON Interface for Power Management Requests and Policies* for more information.

Performance Considerations

While the Haswell microarchitecture allows for independent power control for each core, earlier microarchitectures do not offer such fine-grained control. When deploying on pre-Haswell platforms, greater care must be taken when selecting which cores are assigned to a VM, for example, a core does not scale down in frequency until all of its siblings are similarly scaled down.

4.43.2 Configuration

BIOS

To use the power management features of the DPDK, you must enable Enhanced Intel SpeedStep® Technology in the platform BIOS. Otherwise, the `sys` file folder `/sys/devices/system/cpu/cpu0/cpufreq` does not exist, and you cannot use CPU frequency-based power management. Refer to the relevant BIOS documentation to determine how to access these settings.

Host Operating System

The DPDK Power Management library can use either the `acpi_cpufreq` or the `intel_pstate` kernel driver for the management of core frequencies. In many cases, the `intel_pstate` driver is the default power management environment.

Should the `acpi-cpufreq driver` be required, the `intel_pstate` module must be disabled, and the `acpi-cpufreq` module loaded in its place.

To disable the `intel_pstate` driver, add the following to the `grub` Linux command line:

```
intel_pstate=disable
```

On reboot, load the `acpi_cpufreq` module:

```
modprobe acpi_cpufreq
```

Hypervisor Channel Configuration

Configure `virtio-serial` channels using `libvirt` XML. The XML structure is as follows:

```
<name>{vm_name}</name>
<controller type='virtio-serial' index='0'>
   <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x0'/>
</controller>
<channel type='unix'>
   <source mode='bind' path='/tmp/powermonitor/{vm_name}.{channel_num}'/>
   <target type='virtio' name='virtio.serial.port.poweragent.{vm_channel_num}'/>
   <address type='virtio-serial' controller='0' bus='0' port='{N}'/>
</channel>
```

Where a single controller of type `virtio-serial` is created, up to 32 channels can be associated with a single controller, and multiple controllers can be specified. The convention is to use the name of the VM in the host path `{vm_name}` and to increment `{channel_num}` for each channel. Likewise, the port value `{N}` must be incremented for each channel.

On the host, for each channel to appear in the path, ensure the creation of the `/tmp/powermonitor/` directory and the assignment of `qemu` permissions:

```
mkdir /tmp/powermonitor/
chown qemu:qemu /tmp/powermonitor
```

Note that files and directories in `/tmp` are generally removed when rebooting the host and you may need to perform the previous steps after each reboot.

The serial device as it appears on a VM is configured with the target element attribute name and must be in the form: `virtio.serial.port.poweragent.{vm_channel_num}`, where `vm_channel_num` is typically the lcore channel to be used in DPDK VM applications.

Each channel on a VM is present at:

`/dev/virtio-ports/virtio.serial.port.poweragent.{vm_channel_num}`

4.43.3 Compiling and Running the Host Application

Compiling the Host Application

For information on compiling the DPDK and sample applications, see see *Compiling the Sample Applications*.

The application is located in the `vm_power_manager` subdirectory.

To build just the `vm_power_manager` application using `make`:

```
export RTE_SDK=/path/to/rte_sdk
export RTE_TARGET=build
cd ${RTE_SDK}/examples/vm_power_manager/
make
```

The resulting binary is `${RTE_SDK}/build/examples/vm_power_manager`.

To build just the `vm_power_manager` application using `meson/ninja`:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}
meson build
cd build
ninja
meson configure -Dexamples=vm_power_manager
ninja
```

The resulting binary is `${RTE_SDK}/build/examples/dpdk-vm_power_manager`.

Running the Host Application

The application does not have any specific command line options other than the EAL options:

```
./build/vm_power_mgr [EAL options]
```

The application requires exactly two cores to run. One core for the CLI and the other for the channel endpoint monitor. For example, to run on cores 0 and 1 on a system with four memory channels, issue the command:

```
./build/vm_power_mgr -l 0-1 -n 4
```

After successful initialization, the VM Power Manager CLI prompt appears:

```
vm_power>
```

Now, it is possible to add virtual machines to the VM Power Manager:

```
vm_power> add_vm {vm_name}
```

When a `{vm_name}` is specified with the `add_vm` command, a lookup is performed with `libvirt` to ensure that the VM exists. `{vm_name}` is a unique identifier to associate channels with a particular VM and for executing operations on a VM within the CLI. VMs do not have to be running to add them.

It is possible to issue several commands from the CLI to manage VMs.

Remove the virtual machine identified by `{vm_name}` from the VM Power Manager using the command:

```
rm_vm {vm_name}
```

Add communication channels for the specified VM using the following command. The `virtio` channels must be enabled in the VM configuration (`qemu/libvirt`) and the associated VM must be active. `{list}` is a comma-separated list of channel numbers to add. Specifying the keyword `all` attempts to add all channels for the VM:

```
 set_pcpu {vm_name} {vcpu} {pcpu}

Enable query of physical core information from a VM:
```

```
set_query {vm_name} enable|disable
```

Manual control and inspection can also be carried in relation CPU frequency scaling:

Get the current frequency for each core specified in the mask:

```
 show_cpu_freq_mask {mask}

Set the current frequency for the cores specified in {core_mask} by scaling each up/down/min/
↪max:
```

```
add_channels {vm_name} {list}|all
```

Enable or disable the communication channels in `{list}` (comma-separated) for the specified VM. Alternatively, replace `list` with the keyword `all`. Disabled channels receive packets on the host. However, the commands they specify are ignored. Set the status to enabled to begin processing requests again:

```
set_channel_status {vm_name} {list}|all enabled|disabled
```

Print to the CLI information on the specified VM. The information lists the number of vCPUs, the pinning to pCPU(s) as a bit mask, along with any communication channels associated with each VM, and the status of each channel:

```
show_vm {vm_name}
```

Set the binding of a virtual CPU on a VM with name `{vm_name}` to the physical CPU mask:

```
set_pcpu_mask {vm_name} {vcpu} {pcpu}
```

Set the binding of the virtual CPU on the VM to the physical CPU:

```
set_pcpu {vm_name} {vcpu} {pcpu}
```

It is also possible to perform manual control and inspection in relation to CPU frequency scaling.

Get the current frequency for each core specified in the mask:

```
show_cpu_freq_mask {mask}
```

Set the current frequency for the cores specified in `{core_mask}` by scaling each up/down/min/max:

```
set_cpu_freq {core_mask} up|down|min|max
```

Get the current frequency for the specified core:

```
show_cpu_freq {core_num}
```

Set the current frequency for the specified core by scaling up/down/min/max:

```
set_cpu_freq {core_num} up|down|min|max
```

Command Line Options for Enabling Out-of-band Branch Ratio Monitoring

There are a couple of command line parameters for enabling the out-of-band monitoring of branch ratios on cores doing busy polling using PMDs as described below:

`--core-list {list of cores}`
: Specify the list of cores to monitor the ratio of branch misses to branch hits. A tightly-polling PMD thread has a very low branch ratio, therefore the core frequency scales down to the minimum allowed value. On receiving packets, the code path changes, causing the branch ratio to increase. When the ratio goes above the ratio threshold, the core frequency scales up to the maximum allowed value.

`--branch-ratio {ratio}`
: Specify a floating-point number that identifies the threshold at which to scale up or down for the given workload. The default branch ratio is 0.01 and needs adjustment for different workloads.

4.43.4 Compiling and Running the Guest Applications

It is possible to use the `l3fwd-power` application (for example) with the `vm_power_manager`.

The distribution also provides a guest CLI for validating the setup.

For both `l3fwd-power` and the guest CLI, the host application must use the `add_channels` command to monitor the channels for the VM. To do this, issue the following commands in the host application:

```
vm_power> add_vm vmname
vm_power> add_channels vmname all
vm_power> set_channel_status vmname all enabled
vm_power> show_vm vmname
```

Compiling the Guest Application

For information on compiling DPDK and the sample applications in general, see *Compiling the Sample Applications*.

For compiling and running the `l3fwd-power` sample application, see *L3 Forwarding with Power Management Sample Application*.

The application is in the `guest_cli` subdirectory under `vm_power_manager`.

To build just the `guest_vm_power_manager` application using `make`, issue the following commands:

```
export RTE_SDK=/path/to/rte_sdk
export RTE_TARGET=build
cd ${RTE_SDK}/examples/vm_power_manager/guest_cli/
make
```

The resulting binary is `${RTE_SDK}/build/examples/guest_cli`.

Note: This sample application conditionally links in the Jansson JSON library. Consequently, if you are using a multilib or cross-compile environment, you may need to set the `PKG_CONFIG_LIBDIR` environmental variable to point to the relevant `pkgconfig` folder so that the correct library is linked in.

For example, if you are building for a 32-bit target, you could find the correct directory using the following find command:

```
# find /usr -type d -name pkgconfig
/usr/lib/i386-linux-gnu/pkgconfig
/usr/lib/x86_64-linux-gnu/pkgconfig
```

Then use:

```
export PKG_CONFIG_LIBDIR=/usr/lib/i386-linux-gnu/pkgconfig
```

You then use the `make` command as normal, which should find the 32-bit version of the library, if it installed. If not, the application builds without the JSON interface functionality.

To build just the `vm_power_manager` application using `meson/ninja`:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}
meson build
cd build
ninja
meson configure -Dexamples=vm_power_manager/guest_cli
ninja
```

The resulting binary is `${RTE_SDK}/build/examples/guest_cli`.

Running the Guest Application

The standard EAL command line parameters are necessary:

```
./build/vm_power_mgr [EAL options] -- [guest options]
```

The guest example uses a channel for each lcore enabled. For example, to run on cores 0, 1, 2 and 3:

```
./build/guest_vm_power_mgr -l 0-3
```

Command Line Options Available When Sending a Policy to the Host

Optionally, there are several command line options for a user who needs to send a power policy to the host application:

--vm-name {name of guest vm}
Allows the user to change the virtual machine name passed down to the host application using the power policy. The default is ubuntu2.

--vcpu-list {list vm cores}
A comma-separated list of cores in the VM that the user wants the host application to monitor. The list of cores in any VM starts at zero, and the host application maps these to the physical cores once the policy passes down to the host. Valid syntax includes individual cores 2,3,4, a range of cores 2-4, or a combination of both 1,3,5-7.

--busy-hours {list of busy hours}
A comma-separated list of hours in which to set the core frequency to the maximum. Valid syntax includes individual hours 2,3,4, a range of hours 2-4, or a combination of both 1,3,5-7. Valid hour values are 0 to 23.

--quiet-hours {list of quiet hours}
A comma-separated list of hours in which to set the core frequency to minimum. Valid syntax includes individual hours 2,3,4, a range of hours 2-4, or a combination of both 1,3,5-7. Valid hour values are 0 to 23.

--policy {policy type}
The type of policy. This can be one of the following values:

- TRAFFIC - Based on incoming traffic rates on the NIC.
- TIME - Uses a busy/quiet hours policy.
- BRANCH_RATIO - Uses branch ratio counters to determine core busyness.
- WORKLOAD - Sets the frequency to low, medium or high based on the received policy setting.

Note: Not all policy types need all parameters. For example, BRANCH_RATIO only needs the vcpu-list parameter.

After successful initialization, the VM Power Manager Guest CLI prompt appears:

```
vm_power(guest)>
```

To change the frequency of an lcore, use a `set_cpu_freq` command similar to the following:

```
set_cpu_freq {core_num} up|down|min|max
```

where, `{core_num}` is the lcore and channel to change frequency by scaling up/down/min/max.

To start an application, configure the power policy, and send it to the host, use a command like the following:

```
./build/guest_vm_power_mgr -l 0-3 -n 4 -- --vm-name=ubuntu --policy=BRANCH_RATIO --vcpu-list=2-
↪4
```

Once the VM Power Manager Guest CLI appears, issuing the 'send_policy now' command will send the policy to the host:

```
send_policy now
```

Once the policy is sent to the host, the host application takes over the power monitoring of the specified cores in the policy.

4.43.5 JSON Interface for Power Management Requests and Policies

In addition to the command line interface for the host command, and a `virtio-serial` interface for VM power policies, there is also a JSON interface through which power commands and policies can be sent.

Note: This functionality adds a dependency on the Jansson library. Install the Jansson development package on the system to avail of the JSON parsing functionality in the app. Issue the `apt-get install libjansson-dev` command to install the development package. The command and package name may be different depending on your operating system. It is worth noting that the app builds successfully if this package is not present, but a warning displays during compilation, and the JSON parsing functionality is not present in the app.

Send a request or policy to the VM Power Manager by simply opening a fifo file at `/tmp/powermonitor/fifo`, writing a JSON string to that file, and closing the file.

The JSON string can be a power management request or a policy, and takes the following format:

```
{"packet_type": {
"pair_1": value,
"pair_2": value
}}
```

The `packet_type` header can contain one of two values, depending on whether a power management request or policy is being sent. The two possible values are `instruction` and `policy` and the expected name-value pairs are different depending on which type is sent.

The pairs are in the format of standard JSON name-value pairs. The value type varies between the different name-value pairs, and may be integers, strings, arrays, and so on. See *JSON Interface Examples* for examples of policies and instructions and *JSON Name-value Pairs* for the supported names and value types.

JSON Interface Examples

The following is an example JSON string that creates a time-profile policy.

```
{"policy": {
"name": "ubuntu",
"command": "create",
"policy_type": "TIME",
"busy_hours":[ 17, 18, 19, 20, 21, 22, 23 ],
"quiet_hours":[ 2, 3, 4, 5, 6 ],
"core_list":[ 11 ]
}}
```

The following is an example JSON string that removes the named policy.

```
{"policy": {
"name": "ubuntu",
"command": "destroy",
}}
```

The following is an example JSON string for a power management request.

```
{"instruction": {
"name": "ubuntu",
"command": "power",
"unit": "SCALE_MAX",
"resource_id": 10
}}
```

To query the available frequences of an lcore, use the query_cpu_freq command. Where {core_num} is the lcore to query. Before using this command, please enable responses via the set_query command on the host.

```
query_cpu_freq {core_num}|all
```

To query the capabilities of an lcore, use the query_cpu_caps command. Where {core_num} is the lcore to query. Before using this command, please enable responses via the set_query command on the host.

```
query_cpu_caps {core_num}|all
```

To start the application and configure the power policy, and send it to the host:

```
./build/guest_vm_power_mgr -l 0-3 -n 4 -- --vm-name=ubuntu --policy=BRANCH_RATIO --vcpu-list=2-
↪4
```

Once the VM Power Manager Guest CLI appears, issuing the 'send_policy now' command will send the policy to the host:

```
send_policy now
```

Once the policy is sent to the host, the host application takes over the power monitoring of the specified cores in the policy.

JSON Name-value Pairs

The following are the name-value pairs supported by the JSON interface:

- *avg_packet_thresh*
- *busy_hours*
- *command*
- *core_list*
- *mac_list*
- *max_packet_thresh*
- *name*
- *policy_type*
- *quiet_hours*
- *resource_id*
- *unit*
- *workload*

avg_packet_thresh

Description
The threshold below which the frequency is set to the minimum value for the TRAFFIC policy. If the traffic rate is above this value and below the maximum value, the frequency is set to medium.

Type
integer

Values
The number of packets below which the TRAFFIC policy applies the minimum frequency, or the medium frequency if between the average and maximum thresholds.

Required
Yes

Example
```
"avg_packet_thresh":  100000
```

busy_hours

Description
The hours of the day in which we scale up the cores for busy times.

Type
array of integers

Values
An array with a list of hour values (0-23).

Required
For the TIME policy only.

Example
```
"busy_hours":[ 17, 18, 19, 20, 21, 22, 23 ]
```

command

Description
The type of packet to send to the VM Power Manager. It is possible to create or destroy a policy or send a direct command to adjust the frequency of a core, as is possible on the command line interface.

Type
string

Values
Possible values are: - CREATE: Create a new policy. - DESTROY: Remove an existing policy. - POWER: Send an immediate command, max, min, and so on.

Required
Yes

Example
```
"command":  "CREATE"
```

core_list

Description
The cores to which to apply a policy.

Type
array of integers

Values
An array with a list of virtual CPUs.

Required
For CREATE/DESTROY policy requests only.

Example
```
"core_list":[ 10, 11 ]
```

mac_list

Description
When the policy is of type TRAFFIC, it is necessary to specify the MAC addresses that the host must monitor.

Type
array of strings

Values
An array with a list of MAC address strings.

Required
For TRAFFIC policy types only.

Example
`"mac_list":[ "de:ad:be:ef:01:01","de:ad:be:ef:01:02" ]`

max_packet_thresh

Description
In a policy of type TRAFFIC, the threshold value above which the frequency is set to a maximum.

Type
integer

Values
The number of packets per interval above which the TRAFFIC policy applies the maximum frequency.

Required
For the TRAFFIC policy only.

Example
`"max_packet_thresh":  500000`

name

Description
The name of the VM or host. Allows the parser to associate the policy with the relevant VM or host OS.

Type
string

Values
Any valid string.

Required
Yes

Example
`"name":  "ubuntu2"`

policy_type

Description
The type of policy to apply. See the `--policy` option description for more information.

Type
string

Values
Possible values are:

- TIME: Time-of-day policy. Scale the frequencies of the relevant cores up/down depending on busy and quiet hours.

- TRAFFIC: Use statistics from the NIC and scale up and down accordingly.
- WORKLOAD: Determine how heavily loaded the cores are and scale up and down accordingly.
- BRANCH_RATIO: An out-of-band policy that looks at the ratio between branch hits and misses on a core and uses that information to determine how much packet processing a core is doing.

Required
For `CREATE` and `DESTROY` policy requests only.

Example
```
"policy_type":  "TIME"
```

quiet_hours

Description
The hours of the day to scale down the cores for quiet times.

Type
array of integers

Values
An array with a list of hour numbers with values in the range 0 to 23.

Required
For the TIME policy only.

Example
```
"quiet_hours":[ 2, 3, 4, 5, 6 ]
```

resource_id

Description
The core to which to apply a power command.

Type
integer

Values
A valid core ID for the VM or host OS.

Required
For the `POWER` instruction only.

Example
```
"resource_id":  10
```

unit

Description
The type of power operation to apply in the command.

Type
string

Values

- SCALE_MAX: Scale the frequency of this core to the maximum.
- SCALE_MIN: Scale the frequency of this core to the minimum.
- SCALE_UP: Scale up the frequency of this core.
- SCALE_DOWN: Scale down the frequency of this core.
- ENABLE_TURBO: Enable Intel® Turbo Boost Technology for this core.
- DISABLE_TURBO: Disable Intel® Turbo Boost Technology for this core.

Required
For the `POWER` instruction only.

Example
```
"unit":  "SCALE_MAX"
```

workload

Description
In a policy of type WORKLOAD, it is necessary to specify how heavy the workload is.

Type
string

Values

- HIGH: Scale the frequency of this core to maximum.
- MEDIUM: Scale the frequency of this core to minimum.
- LOW: Scale up the frequency of this core.

Required
For the `WORKLOAD` policy only.

Example
```
"workload":  "MEDIUM"
```

4.44 TEP termination Sample Application

The TEP (Tunnel End point) termination sample application simulates a VXLAN Tunnel Endpoint (VTEP) termination in DPDK, which is used to demonstrate the offload and filtering capabilities of Intel® XL710 10/40 Gigabit Ethernet Controller for VXLAN packet. This sample uses the basic virtio devices management mechanism from vhost example, and also uses the us-vHost interface and tunnel filtering mechanism to direct a specified traffic to a specific VM. In addition, this sample is also designed to show how tunneling protocols can be handled.

4.44.1 Background

With virtualization, overlay networks allow a network structure to be built or imposed across physical nodes which is abstracted away from the actual underlining physical network connections. This allows network isolation, QOS, etc to be provided on a per client basis.

Fig. 4.20: Overlay Networking.

In a typical setup, the network overlay tunnel is terminated at the Virtual/Tunnel End Point (VEP/TEP). The TEP is normally located at the physical host level ideally in the software switch. Due to processing constraints and the inevitable bottleneck that the switch becomes, the ability to offload overlay support features becomes an important requirement. Intel® XL710 10/40 Gigabit Ethernet network card provides hardware filtering and offload capabilities to support overlay networks implementations such as MAC in UDP and MAC in GRE.

4.44.2 Sample Code Overview

The DPDK TEP termination sample code demonstrates the offload and filtering capabilities of Intel® XL710 10/40 Gigabit Ethernet Controller for VXLAN packet.

The sample code is based on vhost library. The vhost library is developed for user space Ethernet switch to easily integrate with vhost functionality.

The sample will support the followings:

- Tunneling packet recognition.
- The port of UDP tunneling is configurable
- Directing incoming traffic to the correct queue based on the tunnel filter type. The supported filter type are listed below.
  - Inner MAC and VLAN and tenant ID
  - Inner MAC and tenant ID, and Outer MAC
  - Inner MAC and tenant ID

  The tenant ID will be assigned from a static internal table based on the us-vhost device ID. Each device will receive a unique device ID. The inner MAC will be learned by the first packet transmitted from a device.
- Decapsulation of RX VXLAN traffic. This is a software only operation.
- Encapsulation of TX VXLAN traffic. This is a software only operation.

- Inner IP and inner L4 checksum offload.
- TSO offload support for tunneling packet.

The following figure shows the framework of the TEP termination sample application based on DPDK vhost lib.

Fig. 4.21: TEP termination Framework Overview

4.44.3 Supported Distributions

The example in this section have been validated with the following distributions:

- Fedora* 18
- Fedora* 19
- Fedora* 20

4.44.4 Compiling the Sample Code

To enable vhost, turn on vhost library in the configure file `config/common_linux`.

```
CONFIG_RTE_LIBRTE_VHOST=y
```

Then following the to compile the sample application shown in *Compiling the Sample Applications*.

4.44.5 Running the Sample Code

1. Go to the examples directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/tep_termination
```

2. Run the tep_termination sample code:

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
                -p 0x1 --dev-basename tep-termination --nb-devices 4
                --udp-port 4789 --filter-type 1
```

Note: Please note the huge-dir parameter instructs the DPDK to allocate its memory from the 2 MB page hugetlbfs.

Parameters

The same parameters with the vhost sample.

Refer to *Parameters* for detailed explanation.

Number of Devices.

The nb-devices option specifies the number of virtIO device. The default value is 2.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
               --nb-devices 2
```

Tunneling UDP port.

The udp-port option is used to specify the destination UDP number for UDP tunneling packet. The default value is 4789.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
               --nb-devices 2 --udp-port 4789
```

Filter Type.

The filter-type option is used to specify which filter type is used to filter UDP tunneling packet to a specified queue. The default value is 1, which means the filter type of inner MAC and tenant ID is used.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
           --nb-devices 2 --udp-port 4789 --filter-type 1
```

TX Checksum.

The tx-checksum option is used to enable or disable the inner header checksum offload. The default value is 0, which means the checksum offload is disabled.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
           --nb-devices 2 --tx-checksum
```

TCP segment size.

The tso-segsz option specifies the TCP segment size for TSO offload for tunneling packet. The default value is 0, which means TSO offload is disabled.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
           --tx-checksum --tso-segsz 800
```

Decapsulation option.

The decap option is used to enable or disable decapsulation operation for received VXLAN packet. The default value is 1.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
           --nb-devices 4 --udp-port 4789 --decap 1
```

Encapsulation option.

The encap option is used to enable or disable encapsulation operation for transmitted packet. The default value is 1.

```
user@target:~$ ./build/app/tep_termination -l 0-3 -n 4 --huge-dir /mnt/huge --
           --nb-devices 4 --udp-port 4789 --encap 1
```

4.44.6 Running the Virtual Machine (QEMU)

Refer to *Start the VM*.

4.44.7 Running DPDK in the Virtual Machine

Refer to *Run testpmd inside guest*.

4.44.8 Passing Traffic to the Virtual Machine Device

For a virtio-net device to receive traffic, the traffic's Layer 2 header must include both the virtio-net device's MAC address. The DPDK sample code behaves in a similar manner to a learning switch in that it learns the MAC address of the virtio-net devices from the first transmitted packet. On learning the MAC address, the DPDK vhost sample code prints a message with the MAC address and tenant ID virtio-net device. For example:

```
DATA: (0) MAC_ADDRESS cc:bb:bb:bb:bb:bb and VNI 1000 registered
```

The above message indicates that device 0 has been registered with MAC address cc:bb:bb:bb:bb:bb and VNI 1000. Any packets received on the NIC with these values are placed on the devices receive queue.

4.45 PTP Client Sample Application

The PTP (Precision Time Protocol) client sample application is a simple example of using the DPDK IEEE1588 API to communicate with a PTP master clock to synchronize the time on the NIC and, optionally, on the Linux system.

Note, PTP is a time syncing protocol and cannot be used within DPDK as a time-stamping mechanism. See the following for an explanation of the protocol: Precision Time Protocol.

4.45.1 Limitations

The PTP sample application is intended as a simple reference implementation of a PTP client using the DPDK IEEE1588 API. In order to keep the application simple the following assumptions are made:

- The first discovered master is the master for the session.
- Only L2 PTP packets are supported.
- Only the PTP v2 protocol is supported.
- Only the slave clock is implemented.

4.45.2 How the Application Works

Fig. 4.22: PTP Synchronization Protocol

The PTP synchronization in the sample application works as follows:

- Master sends *Sync* message - the slave saves it as T2.
- Master sends *Follow Up* message and sends time of T1.
- Slave sends *Delay Request* frame to PTP Master and stores T3.
- Master sends *Delay Response* T4 time which is time of received T3.

The adjustment for slave can be represented as:

adj = -[(T2-T1)-(T4 - T3)]/2

If the command line parameter `-T 1` is used the application also synchronizes the PTP PHC clock with the Linux kernel clock.

4.45.3 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ptpclient` sub-directory.

Note: To compile the application edit the `config/common_linux` configuration file to enable IEEE1588 and then recompile DPDK:

```
CONFIG_RTE_LIBRTE_IEEE1588=y
```

4.45.4 Running the Application

To run the example in a `linux` environment:

```
./build/ptpclient -l 1 -n 4 -- -p 0x1 -T 0
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

- `-p portmask`: Hexadecimal portmask.
- `-T 0`: Update only the PTP slave clock.
- `-T 1`: Update the PTP slave clock and synchronize the Linux Kernel to the PTP clock.

4.45.5 Code Explanation

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with `rte_` and are explained in detail in the *DPDK API Documentation*.

The Main Function

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

And than we parse application specific arguments

```
argc -= ret;
argv += ret;

ret = ptp_parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with PTP initialization\n");
```

The `main()` also allocates a mempool to hold the mbufs (Message Buffers) used by the application:

```
mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
       MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());
```

Mbufs are the packet buffer structure used by DPDK. They are explained in detail in the "Mbuf Library" section of the *DPDK Programmer's Guide*.

The `main()` function also initializes all the ports using the user defined `port_init()` function with portmask provided by user:

```
for (portid = 0; portid < nb_ports; portid++)
    if ((ptp_enabled_port_mask & (1 << portid)) != 0) {

        if (port_init(portid, mbuf_pool) == 0) {
            ptp_enabled_ports[ptp_enabled_port_nb] = portid;
            ptp_enabled_port_nb++;
        } else {
            rte_exit(EXIT_FAILURE, "Cannot init port %"PRIu8 "\n",
                    portid);
        }
    }
```

Once the initialization is complete, the application is ready to launch a function on an lcore. In this example `lcore_main()` is called on a single lcore.

```
lcore_main();
```

The `lcore_main()` function is explained below.

The Lcores Main

As we saw above the `main()` function calls an application function on the available lcores.

The main work of the application is done within the loop:

```
for (portid = 0; portid < ptp_enabled_port_nb; portid++) {

    portid = ptp_enabled_ports[portid];
    nb_rx = rte_eth_rx_burst(portid, 0, &m, 1);

    if (likely(nb_rx == 0))
        continue;

    if (m->ol_flags & PKT_RX_IEEE1588_PTP)
        parse_ptp_frames(portid, m);

    rte_pktmbuf_free(m);
}
```

Packets are received one by one on the RX ports and, if required, PTP response packets are transmitted on the TX ports.

If the offload flags in the mbuf indicate that the packet is a PTP packet then the packet is parsed to determine which type:

```
if (m->ol_flags & PKT_RX_IEEE1588_PTP)
     parse_ptp_frames(portid, m);
```

All packets are freed explicitly using `rte_pktmbuf_free()`.

The forwarding loop can be interrupted and the application closed using `Ctrl-C`.

PTP parsing

The `parse_ptp_frames()` function processes PTP packets, implementing slave PTP IEEE1588 L2 functionality.

```
void
parse_ptp_frames(uint16_t portid, struct rte_mbuf *m) {
    struct ptp_header *ptp_hdr;
    struct rte_ether_hdr *eth_hdr;
    uint16_t eth_type;

    eth_hdr = rte_pktmbuf_mtod(m, struct rte_ether_hdr *);
    eth_type = rte_be_to_cpu_16(eth_hdr->ether_type);

    if (eth_type == PTP_PROTOCOL) {
        ptp_data.m = m;
        ptp_data.portid = portid;
        ptp_hdr = (struct ptp_header *)(rte_pktmbuf_mtod(m, char *)
                    + sizeof(struct rte_ether_hdr));

        switch (ptp_hdr->msgtype) {
        case SYNC:
            parse_sync(&ptp_data);
            break;
        case FOLLOW_UP:
            parse_fup(&ptp_data);
```

(continues on next page)

(continued from previous page)

```
            break;
        case DELAY_RESP:
            parse_drsp(&ptp_data);
            print_clock_info(&ptp_data);
            break;
        default:
            break;
        }
    }
}
```

There are 3 types of packets on the RX path which we must parse to create a minimal implementation of the PTP slave client:

- SYNC packet.
- FOLLOW UP packet
- DELAY RESPONSE packet.

When we parse the *FOLLOW UP* packet we also create and send a *DELAY_REQUEST* packet. Also when we parse the *DELAY RESPONSE* packet, and all conditions are met we adjust the PTP slave clock.

4.46 Performance Thread Sample Application

The performance thread sample application is a derivative of the standard L3 forwarding application that demonstrates different threading models.

4.46.1 Overview

For a general description of the L3 forwarding applications capabilities please refer to the documentation of the standard application in *L3 Forwarding Sample Application*.

The performance thread sample application differs from the standard L3 forwarding example in that it divides the TX and RX processing between different threads, and makes it possible to assign individual threads to different cores.

Three threading models are considered:

1. When there is one EAL thread per physical core.
2. When there are multiple EAL threads per physical core.
3. When there are multiple lightweight threads per EAL thread.

Since DPDK release 2.0 it is possible to launch applications using the `--lcores` EAL parameter, specifying cpu-sets for a physical core. With the performance thread sample application its is now also possible to assign individual RX and TX functions to different cores.

As an alternative to dividing the L3 forwarding work between different EAL threads the performance thread sample introduces the possibility to run the application threads as lightweight threads (L-threads) within one or more EAL threads.

In order to facilitate this threading model the example includes a primitive cooperative scheduler (L-thread) subsystem. More details of the L-thread subsystem can be found in *The L-thread subsystem*.

Note: Whilst theoretically possible it is not anticipated that multiple L-thread schedulers would be run on the same physical core, this mode of operation should not be expected to yield useful performance and is considered invalid.

4.46.2 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the *performance-thread/l3fwd-thread* sub-directory.

4.46.3 Running the Application

The application has a number of command line options:

```
./build/l3fwd-thread [EAL options] --
    -p PORTMASK [-P]
    --rx(port,queue,lcore,thread)[,(port,queue,lcore,thread)]
    --tx(lcore,thread)[,(lcore,thread)]
    [--enable-jumbo] [--max-pkt-len PKTLEN]]  [--no-numa]
    [--hash-entry-num] [--ipv6] [--no-lthreads] [--stat-lcore lcore]
    [--parse-ptype]
```

Where:

- `-p PORTMASK`: Hexadecimal bitmask of ports to configure.
- `-P`: optional, sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- `--rx (port,queue,lcore,thread)[,(port,queue,lcore,thread)]`: the list of NIC RX ports and queues handled by the RX lcores and threads. The parameters are explained below.
- `--tx (lcore,thread)[,(lcore,thread)]`: the list of TX threads identifying the lcore the thread runs on, and the id of RX thread with which it is associated. The parameters are explained below.
- `--enable-jumbo`: optional, enables jumbo frames.
- `--max-pkt-len`: optional, maximum packet length in decimal (64-9600).
- `--no-numa`: optional, disables numa awareness.
- `--hash-entry-num`: optional, specifies the hash entry number in hex to be setup.
- `--ipv6`: optional, set it if running ipv6 packets.
- `--no-lthreads`: optional, disables l-thread model and uses EAL threading model. See below.
- `--stat-lcore`: optional, run CPU load stats collector on the specified lcore.
- `--parse-ptype:` optional, set to use software to analyze packet type. Without this option, hardware will check the packet type.

The parameters of the `--rx` and `--tx` options are:

- `--rx` parameters

port	RX port
queue	RX queue that will be read on the specified RX port
lcore	Core to use for the thread
thread	Thread id (continuously from 0 to N)

- `--tx` parameters

lcore	Core to use for L3 route match and transmit
thread	Id of RX thread to be associated with this TX thread

The `l3fwd-thread` application allows you to start packet processing in two threading models: L-Threads (default) and EAL Threads (when the `--no-lthreads` parameter is used). For consistency all parameters are used in the same way for both models.

Running with L-threads

When the L-thread model is used (default option), lcore and thread parameters in `--rx/--tx` are used to affinitize threads to the selected scheduler.

For example, the following places every l-thread on different lcores:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)"
```

The following places RX l-threads on lcore 0 and TX l-threads on lcore 1 and 2 and so on:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,0,1)" \
             --tx="(1,0)(2,1)"
```

Running with EAL threads

When the `--no-lthreads` parameter is used, the L-threading model is turned off and EAL threads are used for all processing. EAL threads are enumerated in the same way as L-threads, but the `--lcores` EAL parameter is used to affinitize threads to the selected cpu-set (scheduler). Thus it is possible to place every RX and TX thread on different lcores.

For example, the following places every EAL thread on different lcores:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

To affinitize two or more EAL threads to one cpu-set, the EAL `--lcores` parameter is used.

The following places RX EAL threads on lcore 0 and TX EAL threads on lcore 1 and 2 and so on:

```
l3fwd-thread -l 0-7 -n 2 --lcores="(0,1)@0,(2,3)@1" -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

Examples

For selected scenarios the command line configuration of the application for L-threads and its corresponding EAL threads command line can be realized as follows:

a) Start every thread on different scheduler (1:1):

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)"
```

EAL thread equivalent:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

b) Start all threads on one core (N:1).

Start 4 L-threads on lcore 0:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,0,1)" \
             --tx="(0,0)(0,1)"
```

Start 4 EAL threads on cpu-set 0:

```
l3fwd-thread -l 0-7 -n 2 --lcores="(0-3)@0" -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,0,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

c) Start threads on different cores (N:M).

Start 2 L-threads for RX on lcore 0, and 2 L-threads for TX on lcore 1:

```
l3fwd-thread -l 0-7 -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,0,1)" \
             --tx="(1,0)(1,1)"
```

Start 2 EAL threads for RX on cpu-set 0, and 2 EAL threads for TX on cpu-set 1:

```
l3fwd-thread -l 0-7 -n 2 --lcores="(0-1)@0,(2-3)@1" -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

4.46.4 Explanation

To a great extent the sample application differs little from the standard L3 forwarding application, and readers are advised to familiarize themselves with the material covered in the *L3 Forwarding Sample Application* documentation before proceeding.

The following explanation is focused on the way threading is handled in the performance thread example.

Mode of operation with EAL threads

The performance thread sample application has split the RX and TX functionality into two different threads, and the RX and TX threads are interconnected via software rings. With respect to these rings the RX threads are producers and the TX threads are consumers.

On initialization the TX and RX threads are started according to the command line parameters.

The RX threads poll the network interface queues and post received packets to a TX thread via a corresponding software ring.

The TX threads poll software rings, perform the L3 forwarding hash/LPM match, and assemble packet bursts before performing burst transmit on the network interface.

As with the standard L3 forward application, burst draining of residual packets is performed periodically with the period calculated from elapsed time using the timestamps counter.

The diagram below illustrates a case with two RX threads and three TX threads.

Mode of operation with L-threads

Like the EAL thread configuration the application has split the RX and TX functionality into different threads, and the pairs of RX and TX threads are interconnected via software rings.

On initialization an L-thread scheduler is started on every EAL thread. On all but the master EAL thread only a dummy L-thread is initially started. The L-thread started on the master EAL thread then spawns other L-threads on different L-thread schedulers according the command line parameters.

The RX threads poll the network interface queues and post received packets to a TX thread via the corresponding software ring.

The ring interface is augmented by means of an L-thread condition variable that enables the TX thread to be suspended when the TX ring is empty. The RX thread signals the condition whenever it posts to the TX ring, causing the TX thread to be resumed.

Additionally the TX L-thread spawns a worker L-thread to take care of polling the software rings, whilst it handles burst draining of the transmit buffer.

The worker threads poll the software rings, perform L3 route lookup and assemble packet bursts. If the TX ring is empty the worker thread suspends itself by waiting on the condition variable associated with the ring.

Burst draining of residual packets, less than the burst size, is performed by the TX thread which sleeps (using an L-thread sleep function) and resumes periodically to flush the TX buffer.

This design means that L-threads that have no work, can yield the CPU to other L-threads and avoid having to constantly poll the software rings.

The diagram below illustrates a case with two RX threads and three TX functions (each comprising a thread that processes forwarding and a thread that periodically drains the output buffer of residual packets).

CPU load statistics

It is possible to display statistics showing estimated CPU load on each core. The statistics indicate the percentage of CPU time spent: processing received packets (forwarding), polling queues/rings (waiting for work), and doing any other processing (context switch and other overhead).

When enabled statistics are gathered by having the application threads set and clear flags when they enter and exit pertinent code sections. The flags are then sampled in real time by a statistics collector thread running on another core. This thread displays the data in real time on the console.

This feature is enabled by designating a statistics collector core, using the `--stat-lcore` parameter.

4.46.5 The L-thread subsystem

The L-thread subsystem resides in the examples/performance-thread/common directory and is built and linked automatically when building the `l3fwd-thread` example.

The subsystem provides a simple cooperative scheduler to enable arbitrary functions to run as cooperative threads within a single EAL thread. The subsystem provides a pthread like API that is intended to assist in reuse of legacy code written for POSIX pthreads.

The following sections provide some detail on the features, constraints, performance and porting considerations when using L-threads.

Comparison between L-threads and POSIX pthreads

The fundamental difference between the L-thread and pthread models is the way in which threads are scheduled. The simplest way to think about this is to consider the case of a processor with a single CPU. To run multiple threads on a single CPU, the scheduler must frequently switch between the threads, in order that each thread is able to make timely progress. This is the basis of any multitasking operating system.

This section explores the differences between the pthread model and the L-thread model as implemented in the provided L-thread subsystem. If needed a theoretical discussion of preemptive vs cooperative multi-threading can be found in any good text on operating system design.

Scheduling and context switching

The POSIX pthread library provides an application programming interface to create and synchronize threads. Scheduling policy is determined by the host OS, and may be configurable. The OS may use sophisticated rules to determine which thread should be run next, threads may suspend themselves or make other threads ready, and the scheduler may employ a time slice giving each thread a maximum time quantum after which it will be preempted in favor of another thread that is ready to run. To complicate matters further threads may be assigned different scheduling priorities.

By contrast the L-thread subsystem is considerably simpler. Logically the L-thread scheduler performs the same multiplexing function for L-threads within a single pthread as the OS scheduler does for pthreads within an application process. The L-thread scheduler is simply the main loop of a pthread, and in so far as the host OS is concerned it is a regular pthread just like any other. The host OS is oblivious about the existence of and not at all involved in the scheduling of L-threads.

The other and most significant difference between the two models is that L-threads are scheduled cooperatively. L-threads cannot not preempt each other, nor can the L-thread scheduler preempt a running

L-thread (i.e. there is no time slicing). The consequence is that programs implemented with L-threads must possess frequent rescheduling points, meaning that they must explicitly and of their own volition return to the scheduler at frequent intervals, in order to allow other L-threads an opportunity to proceed.

In both models switching between threads requires that the current CPU context is saved and a new context (belonging to the next thread ready to run) is restored. With pthreads this context switching is handled transparently and the set of CPU registers that must be preserved between context switches is as per an interrupt handler.

An L-thread context switch is achieved by the thread itself making a function call to the L-thread scheduler. Thus it is only necessary to preserve the callee registers. The caller is responsible to save and restore any other registers it is using before a function call, and restore them on return, and this is handled by the compiler. For `X86_64` on both Linux and BSD the System V calling convention is used, this defines registers RSP, RBP, and R12-R15 as callee-save registers (for more detailed discussion a good reference is X86 Calling Conventions).

Taking advantage of this, and due to the absence of preemption, an L-thread context switch is achieved with less than 20 load/store instructions.

The scheduling policy for L-threads is fixed, there is no prioritization of L-threads, all L-threads are equal and scheduling is based on a FIFO ready queue.

An L-thread is a struct containing the CPU context of the thread (saved on context switch) and other useful items. The ready queue contains pointers to threads that are ready to run. The L-thread scheduler is a simple loop that polls the ready queue, reads from it the next thread ready to run, which it resumes by saving the current context (the current position in the scheduler loop) and restoring the context of the next thread from its thread struct. Thus an L-thread is always resumed at the last place it yielded.

A well behaved L-thread will call the context switch regularly (at least once in its main loop) thus returning to the scheduler's own main loop. Yielding inserts the current thread at the back of the ready queue, and the process of servicing the ready queue is repeated, thus the system runs by flipping back and forth the between L-threads and scheduler loop.

In the case of pthreads, the preemptive scheduling, time slicing, and support for thread prioritization means that progress is normally possible for any thread that is ready to run. This comes at the price of a relatively heavier context switch and scheduling overhead.

With L-threads the progress of any particular thread is determined by the frequency of rescheduling opportunities in the other L-threads. This means that an errant L-thread monopolizing the CPU might cause scheduling of other threads to be stalled. Due to the lower cost of context switching, however, voluntary rescheduling to ensure progress of other threads, if managed sensibly, is not a prohibitive overhead, and overall performance can exceed that of an application using pthreads.

Mutual exclusion

With pthreads preemption means that threads that share data must observe some form of mutual exclusion protocol.

The fact that L-threads cannot preempt each other means that in many cases mutual exclusion devices can be completely avoided.

Locking to protect shared data can be a significant bottleneck in multi-threaded applications so a carefully designed cooperatively scheduled program can enjoy significant performance advantages.

So far we have considered only the simplistic case of a single core CPU, when multiple CPUs are considered things are somewhat more complex.

First of all it is inevitable that there must be multiple L-thread schedulers, one running on each EAL thread. So long as these schedulers remain isolated from each other the above assertions about the potential advantages of cooperative scheduling hold true.

A configuration with isolated cooperative schedulers is less flexible than the pthread model where threads can be affinitized to run on any CPU. With isolated schedulers scaling of applications to utilize fewer or more CPUs according to system demand is very difficult to achieve.

The L-thread subsystem makes it possible for L-threads to migrate between schedulers running on different CPUs. Needless to say if the migration means that threads that share data end up running on different CPUs then this will introduce the need for some kind of mutual exclusion system.

Of course `rte_ring` software rings can always be used to interconnect threads running on different cores, however to protect other kinds of shared data structures, lock free constructs or else explicit locking will be required. This is a consideration for the application design.

In support of this extended functionality, the L-thread subsystem implements thread safe mutexes and condition variables.

The cost of affinitizing and of condition variable signaling is significantly lower than the equivalent pthread operations, and so applications using these features will see a performance benefit.

Thread local storage

As with applications written for pthreads an application written for L-threads can take advantage of thread local storage, in this case local to an L-thread. An application may save and retrieve a single pointer to application data in the L-thread struct.

For legacy and backward compatibility reasons two alternative methods are also offered, the first is modeled directly on the pthread get/set specific APIs, the second approach is modeled on the `RTE_PER_LCORE` macros, whereby `PER_LTHREAD` macros are introduced, in both cases the storage is local to the L-thread.

Constraints and performance implications when using L-threads

API compatibility

The L-thread subsystem provides a set of functions that are logically equivalent to the corresponding functions offered by the POSIX pthread library, however not all pthread functions have a corresponding L-thread equivalent, and not all features available to pthreads are implemented for L-threads.

The pthread library offers considerable flexibility via programmable attributes that can be associated with threads, mutexes, and condition variables.

By contrast the L-thread subsystem has fixed functionality, the scheduler policy cannot be varied, and L-threads cannot be prioritized. There are no variable attributes associated with any L-thread objects. L-threads, mutexes and conditional variables, all have fixed functionality. (Note: reserved parameters are included in the APIs to facilitate possible future support for attributes).

The table below lists the pthread and equivalent L-thread APIs with notes on differences and/or constraints. Where there is no L-thread entry in the table, then the L-thread subsystem provides no equivalent function.

Table 4.5: Pthread and equivalent L-thread APIs.

Pthread function	L-thread function	Notes
pthread_barrier_destroy		
pthread_barrier_init		
pthread_barrier_wait		
pthread_cond_broadcast	lthread_cond_broadcast	See note 1
pthread_cond_destroy	lthread_cond_destroy	
pthread_cond_init	lthread_cond_init	
pthread_cond_signal	lthread_cond_signal	See note 1
pthread_cond_timedwait		
pthread_cond_wait	lthread_cond_wait	See note 5
pthread_create	lthread_create	See notes 2, 3
pthread_detach	lthread_detach	See note 4
pthread_equal		
pthread_exit	lthread_exit	
pthread_getspecific	lthread_getspecific	
pthread_getcpuclockid		
pthread_join	lthread_join	
pthread_key_create	lthread_key_create	
pthread_key_delete	lthread_key_delete	
pthread_mutex_destroy	lthread_mutex_destroy	
pthread_mutex_init	lthread_mutex_init	
pthread_mutex_lock	lthread_mutex_lock	See note 6
pthread_mutex_trylock	lthread_mutex_trylock	See note 6
pthread_mutex_timedlock		
pthread_mutex_unlock	lthread_mutex_unlock	
pthread_once		
pthread_rwlock_destroy		
pthread_rwlock_init		
pthread_rwlock_rdlock		
pthread_rwlock_timedrdlock		
pthread_rwlock_timedwrlock		
pthread_rwlock_tryrdlock		
pthread_rwlock_trywrlock		
pthread_rwlock_unlock		
pthread_rwlock_wrlock		
pthread_self	lthread_current	
pthread_setspecific	lthread_setspecific	
pthread_spin_init		See note 10
pthread_spin_destroy		See note 10
pthread_spin_lock		See note 10
pthread_spin_trylock		See note 10
pthread_spin_unlock		See note 10
pthread_cancel	lthread_cancel	
pthread_setcancelstate		
pthread_setcanceltype		
pthread_testcancel		
pthread_getschedparam		
pthread_setschedparam		

continues on next page

Table 4.5 – continued from previous page

Pthread function	L-thread function	Notes
pthread_yield	lthread_yield	See note 7
pthread_setaffinity_np	lthread_set_affinity	See notes 2, 3, 8
	lthread_sleep	See note 9
	lthread_sleep_clks	See note 9

Note 1:

Neither lthread signal nor broadcast may be called concurrently by L-threads running on different schedulers, although multiple L-threads running in the same scheduler may freely perform signal or broadcast operations. L-threads running on the same or different schedulers may always safely wait on a condition variable.

Note 2:

Pthread attributes may be used to affinitize a pthread with a cpu-set. The L-thread subsystem does not support a cpu-set. An L-thread may be affinitized only with a single CPU at any time.

Note 3:

If an L-thread is intended to run on a different NUMA node than the node that creates the thread then, when calling `lthread_create()` it is advantageous to specify the destination core as a parameter of `lthread_create()`. See *Memory allocation and NUMA awareness* for details.

Note 4:

An L-thread can only detach itself, and cannot detach other L-threads.

Note 5:

A wait operation on a pthread condition variable is always associated with and protected by a mutex which must be owned by the thread at the time it invokes `pthread_wait()`. By contrast L-thread condition variables are thread safe (for waiters) and do not use an associated mutex. Multiple L-threads (including L-threads running on other schedulers) can safely wait on a L-thread condition variable. As a consequence the performance of an L-thread condition variables is typically an order of magnitude faster than its pthread counterpart.

Note 6:

Recursive locking is not supported with L-threads, attempts to take a lock recursively will be detected and rejected.

Note 7:

`lthread_yield()` will save the current context, insert the current thread to the back of the ready queue, and resume the next ready thread. Yielding increases ready queue backlog, see *Ready queue backlog* for more details about the implications of this.

N.B. The context switch time as measured from immediately before the call to `lthread_yield()` to the point at which the next ready thread is resumed, can be an order of magnitude faster that the same measurement for pthread_yield.

Note 8:

`lthread_set_affinity()` is similar to a yield apart from the fact that the yielding thread is inserted into a peer ready queue of another scheduler. The peer ready queue is actually a separate thread safe queue, which means that threads appearing in the peer ready queue can jump any backlog in the local ready queue on the destination scheduler.

The context switch time as measured from the time just before the call to `lthread_set_affinity()` to just after the same thread is resumed on the new scheduler can be orders of magnitude faster than the same measurement for `pthread_setaffinity_np()`.

Note 9:

Although there is no `pthread_sleep()` function, `lthread_sleep()` and `lthread_sleep_clks()` can be used wherever `sleep()`, `usleep()` or `nanosleep()` might ordinarily be used. The L-thread sleep functions suspend the current thread, start an `rte_timer` and resume the thread when the timer matures. The `rte_timer_manage()` entry point is called on every pass of the scheduler loop. This means that the worst case jitter on timer expiry is determined by the longest period between context switches of any running L-threads.

In a synthetic test with many threads sleeping and resuming then the measured jitter is typically orders of magnitude lower than the same measurement made for `nanosleep()`.

Note 10:

Spin locks are not provided because they are problematical in a cooperative environment, see *Locks and spinlocks* for a more detailed discussion on how to avoid spin locks.

Thread local storage

Of the three L-thread local storage options the simplest and most efficient is storing a single application data pointer in the L-thread struct.

The `PER_LTHREAD` macros involve a run time computation to obtain the address of the variable being saved/retrieved and also require that the accesses are de-referenced via a pointer. This means that code that has used `RTE_PER_LCORE` macros being ported to L-threads might need some slight adjustment (see *Thread local storage* for hints about porting code that makes use of thread local storage).

The get/set specific APIs are consistent with their pthread counterparts both in use and in performance.

Memory allocation and NUMA awareness

All memory allocation is from DPDK huge pages, and is NUMA aware. Each scheduler maintains its own caches of objects: lthreads, their stacks, TLS, mutexes and condition variables. These caches are implemented as unbounded lock free MPSC queues. When objects are created they are always allocated from the caches on the local core (current EAL thread).

If an L-thread has been affinitized to a different scheduler, then it can always safely free resources to the caches from which they originated (because the caches are MPSC queues).

If the L-thread has been affinitized to a different NUMA node then the memory resources associated with it may incur longer access latency.

The commonly used pattern of setting affinity on entry to a thread after it has started, means that memory allocation for both the stack and TLS will have been made from caches on the NUMA node on which the threads creator is running. This has the side effect that access latency will be sub-optimal after affinitizing.

This side effect can be mitigated to some extent (although not completely) by specifying the destination CPU as a parameter of `lthread_create()` this causes the L-thread's stack and TLS to be allocated when it is first scheduled on the destination scheduler, if the destination is a on another NUMA node it results in a more optimal memory allocation.

Note that the lthread struct itself remains allocated from memory on the creating node, this is unavoidable because an L-thread is known everywhere by the address of this struct.

Object cache sizing

The per lcore object caches pre-allocate objects in bulk whenever a request to allocate an object finds a cache empty. By default 100 objects are pre-allocated, this is defined by `LTHREAD_PREALLOC` in the public API header file lthread_api.h. This means that the caches constantly grow to meet system demand.

In the present implementation there is no mechanism to reduce the cache sizes if system demand reduces. Thus the caches will remain at their maximum extent indefinitely.

A consequence of the bulk pre-allocation of objects is that every 100 (default value) additional new object create operations results in a call to `rte_malloc()`. For creation of objects such as L-threads, which trigger the allocation of even more objects (i.e. their stacks and TLS) then this can cause outliers in scheduling performance.

If this is a problem the simplest mitigation strategy is to dimension the system, by setting the bulk object pre-allocation size to some large number that you do not expect to be exceeded. This means the caches will be populated once only, the very first time a thread is created.

Ready queue backlog

One of the more subtle performance considerations is managing the ready queue backlog. The fewer threads that are waiting in the ready queue then the faster any particular thread will get serviced.

In a naive L-thread application with N L-threads simply looping and yielding, this backlog will always be equal to the number of L-threads, thus the cost of a yield to a particular L-thread will be N times the context switch time.

This side effect can be mitigated by arranging for threads to be suspended and wait to be resumed, rather than polling for work by constantly yielding. Blocking on a mutex or condition variable or even more obviously having a thread sleep if it has a low frequency workload are all mechanisms by which a thread can be excluded from the ready queue until it really does need to be run. This can have a significant positive impact on performance.

Initialization, shutdown and dependencies

The L-thread subsystem depends on DPDK for huge page allocation and depends on the `rte_timer subsystem`. The DPDK EAL initialization and `rte_timer_subsystem_init()` **MUST** be completed before the L-thread sub system can be used.

Thereafter initialization of the L-thread subsystem is largely transparent to the application. Constructor functions ensure that global variables are properly initialized. Other than global variables each scheduler is initialized independently the first time that an L-thread is created by a particular EAL thread.

If the schedulers are to be run as isolated and independent schedulers, with no intention that L-threads running on different schedulers will migrate between schedulers or synchronize with L-threads running on other schedulers, then initialization consists simply of creating an L-thread, and then running the L-thread scheduler.

If there will be interaction between L-threads running on different schedulers, then it is important that the starting of schedulers on different EAL threads is synchronized.

To achieve this an additional initialization step is necessary, this is simply to set the number of schedulers by calling the API function `lthread_num_schedulers_set(n)`, where `n` is the number of EAL threads that will run L-thread schedulers. Setting the number of schedulers to a number greater than 0 will cause all schedulers to wait until the others have started before beginning to schedule L-threads.

The L-thread scheduler is started by calling the function `lthread_run()` and should be called from the EAL thread and thus become the main loop of the EAL thread.

The function `lthread_run()`, will not return until all threads running on the scheduler have exited, and the scheduler has been explicitly stopped by calling `lthread_scheduler_shutdown(lcore)` or `lthread_scheduler_shutdown_all()`.

All these function do is tell the scheduler that it can exit when there are no longer any running L-threads, neither function forces any running L-thread to terminate. Any desired application shutdown behavior must be designed and built into the application to ensure that L-threads complete in a timely manner.

Important Note: It is assumed when the scheduler exits that the application is terminating for good, the scheduler does not free resources before exiting and running the scheduler a subsequent time will result in undefined behavior.

Porting legacy code to run on L-threads

Legacy code originally written for a pthread environment may be ported to L-threads if the considerations about differences in scheduling policy, and constraints discussed in the previous sections can be accommodated.

This section looks in more detail at some of the issues that may have to be resolved when porting code.

pthread API compatibility

The first step is to establish exactly which pthread APIs the legacy application uses, and to understand the requirements of those APIs. If there are corresponding L-lthread APIs, and where the default pthread functionality is used by the application then, notwithstanding the other issues discussed here, it should be feasible to run the application with L-threads. If the legacy code modifies the default behavior using attributes then if may be necessary to make some adjustments to eliminate those requirements.

Blocking system API calls

It is important to understand what other system services the application may be using, bearing in mind that in a cooperatively scheduled environment a thread cannot block without stalling the scheduler and with it all other cooperative threads. Any kind of blocking system call, for example file or socket IO, is a potential problem, a good tool to analyze the application for this purpose is the `strace` utility.

There are many strategies to resolve these kind of issues, each with it merits. Possible solutions include:

- Adopting a polled mode of the system API concerned (if available).
- Arranging for another core to perform the function and synchronizing with that core via constructs that will not block the L-thread.
- Affinitizing the thread to another scheduler devoted (as a matter of policy) to handling threads wishing to make blocking calls, and then back again when finished.

Locks and spinlocks

Locks and spinlocks are another source of blocking behavior that for the same reasons as system calls will need to be addressed.

If the application design ensures that the contending L-threads will always run on the same scheduler then it its probably safe to remove locks and spin locks completely.

The only exception to the above rule is if for some reason the code performs any kind of context switch whilst holding the lock (e.g. yield, sleep, or block on a different lock, or on a condition variable). This will need to determined before deciding to eliminate a lock.

If a lock cannot be eliminated then an L-thread mutex can be substituted for either kind of lock.

An L-thread blocking on an L-thread mutex will be suspended and will cause another ready L-thread to be resumed, thus not blocking the scheduler. When default behavior is required, it can be used as a direct replacement for a pthread mutex lock.

Spin locks are typically used when lock contention is likely to be rare and where the period during which the lock may be held is relatively short. When the contending L-threads are running on the same scheduler then an L-thread blocking on a spin lock will enter an infinite loop stopping the scheduler completely (see *Infinite loops* below).

If the application design ensures that contending L-threads will always run on different schedulers then it might be reasonable to leave a short spin lock that rarely experiences contention in place.

If after all considerations it appears that a spin lock can neither be eliminated completely, replaced with an L-thread mutex, or left in place as is, then an alternative is to loop on a flag, with a call to `lthread_yield()` inside the loop (n.b. if the contending L-threads might ever run on different schedulers the flag will need to be manipulated atomically).

Spinning and yielding is the least preferred solution since it introduces ready queue backlog (see also *Ready queue backlog*).

Sleeps and delays

Yet another kind of blocking behavior (albeit momentary) are delay functions like `sleep()`, `usleep()`, `nanosleep()` etc. All will have the consequence of stalling the L-thread scheduler and unless the delay is very short (e.g. a very short nanosleep) calls to these functions will need to be eliminated.

The simplest mitigation strategy is to use the L-thread sleep API functions, of which two variants exist, `lthread_sleep()` and `lthread_sleep_clks()`. These functions start an rte_timer against the L-thread, suspend the L-thread and cause another ready L-thread to be resumed. The suspended L-thread is resumed when the rte_timer matures.

Infinite loops

Some applications have threads with loops that contain no inherent rescheduling opportunity, and rely solely on the OS time slicing to share the CPU. In a cooperative environment this will stop everything dead. These kind of loops are not hard to identify, in a debug session you will find the debugger is always stopping in the same loop.

The simplest solution to this kind of problem is to insert an explicit `lthread_yield()` or `lthread_sleep()` into the loop. Another solution might be to include the function performed by the loop into the execution path of some other loop that does in fact yield, if this is possible.

Thread local storage

If the application uses thread local storage, the use case should be studied carefully.

In a legacy pthread application either or both the `__thread` prefix, or the pthread set/get specific APIs may have been used to define storage local to a pthread.

In some applications it may be a reasonable assumption that the data could or in fact most likely should be placed in L-thread local storage.

If the application (like many DPDK applications) has assumed a certain relationship between a pthread and the CPU to which it is affinitized, there is a risk that thread local storage may have been used to save some data items that are correctly logically associated with the CPU, and others items which relate to application context for the thread. Only a good understanding of the application will reveal such cases.

If the application requires an that an L-thread is to be able to move between schedulers then care should be taken to separate these kinds of data, into per lcore, and per L-thread storage. In this way a migrating thread will bring with it the local data it needs, and pick up the new logical core specific values from pthread local storage at its new home.

Pthread shim

A convenient way to get something working with legacy code can be to use a shim that adapts pthread API calls to the corresponding L-thread ones. This approach will not mitigate any of the porting considerations mentioned in the previous sections, but it will reduce the amount of code churn that would otherwise been involved. It is a reasonable approach to evaluate L-threads, before investing effort in porting to the native L-thread APIs.

Overview

The L-thread subsystem includes an example pthread shim. This is a partial implementation but does contain the API stubs needed to get basic applications running. There is a simple "hello world" application that demonstrates the use of the pthread shim.

A subtlety of working with a shim is that the application will still need to make use of the genuine pthread library functions, at the very least in order to create the EAL threads in which the L-thread schedulers will run. This is the case with DPDK initialization, and exit.

To deal with the initialization and shutdown scenarios, the shim is capable of switching on or off its adaptor functionality, an application can control this behavior by the calling the function `pt_override_set()`. The default state is disabled.

The pthread shim uses the dynamic linker loader and saves the loaded addresses of the genuine pthread API functions in an internal table, when the shim functionality is enabled it performs the adaptor function, when disabled it invokes the genuine pthread function.

The function `pthread_exit()` has additional special handling. The standard system header file pthread.h declares `pthread_exit()` with `__rte_noreturn` this is an optimization that is possible because the pthread is terminating and this enables the compiler to omit the normal handling of stack and protection of registers since the function is not expected to return, and in fact the thread is being destroyed. These optimizations are applied in both the callee and the caller of the `pthread_exit()` function.

In our cooperative scheduling environment this behavior is inadmissible. The pthread is the L-thread scheduler thread, and, although an L-thread is terminating, there must be a return to the scheduler in order that the system can continue to run. Further, returning from a function with attribute `noreturn` is invalid and may result in undefined behavior.

The solution is to redefine the `pthread_exit` function with a macro, causing it to be mapped to a stub function in the shim that does not have the `noreturn` attribute. This macro is defined in the file `pthread_shim.h`. The stub function is otherwise no different than any of the other stub functions in the shim, and will switch between the real `pthread_exit()` function or the `lthread_exit()` function as required. The only difference is that the mapping to the stub by macro substitution.

A consequence of this is that the file `pthread_shim.h` must be included in legacy code wishing to make use of the shim. It also means that dynamic linkage of a pre-compiled binary that did not include pthread_shim.h is not be supported.

Given the requirements for porting legacy code outlined in *Porting legacy code to run on L-threads* most applications will require at least some minimal adjustment and recompilation to run on L-threads so pre-compiled binaries are unlikely to be met in practice.

In summary the shim approach adds some overhead but can be a useful tool to help establish the feasibility of a code reuse project. It is also a fairly straightforward task to extend the shim if necessary.

Note: Bearing in mind the preceding discussions about the impact of making blocking calls then switching the shim in and out on the fly to invoke any pthread API this might block is something that should typically be avoided.

Building and running the pthread shim

The shim example application is located in the sample application in the performance-thread folder

To build and run the pthread shim example

1. Go to the example applications folder

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/performance-thread/pthread_shim
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linux-gcc
```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

3. Build the application:

```
make
```

4. To run the pthread_shim example

```
lthread-pthread-shim -c core_mask -n number_of_channels
```

L-thread Diagnostics

When debugging you must take account of the fact that the L-threads are run in a single pthread. The current scheduler is defined by `RTE_PER_LCORE(this_sched)`, and the current lthread is stored at `RTE_PER_LCORE(this_sched)->current_lthread`. Thus on a breakpoint in a GDB session the current lthread can be obtained by displaying the pthread local variable `per_lcore_this_sched->current_lthread`.

Another useful diagnostic feature is the possibility to trace significant events in the life of an L-thread, this feature is enabled by changing the value of LTHREAD_DIAG from 0 to 1 in the file `lthread_diag_api.h`.

Tracing of events can be individually masked, and the mask may be programmed at run time. An unmasked event results in a callback that provides information about the event. The default callback simply prints trace information. The default mask is 0 (all events off) the mask can be modified by calling the function `lthread_diagniostic_set_mask()`.

It is possible register a user callback function to implement more sophisticated diagnostic functions. Object creation events (lthread, mutex, and condition variable) accept, and store in the created object, a user supplied reference value returned by the callback function.

The lthread reference value is passed back in all subsequent event callbacks, the mutex and APIs are provided to retrieve the reference value from mutexes and condition variables. This enables a user to monitor, count, or filter for specific events, on specific objects, for example to monitor for a specific thread signaling a specific condition variable, or to monitor on all timer events, the possibilities and combinations are endless.

The callback function can be set by calling the function `lthread_diagnostic_enable()` supplying a callback function pointer and an event mask.

Setting `LTHREAD_DIAG` also enables counting of statistics about cache and queue usage, and these statistics can be displayed by calling the function `lthread_diag_stats_display()`. This function also performs a consistency check on the caches and queues. The function should only be called from the master EAL thread after all slave threads have stopped and returned to the C main program, otherwise the consistency check will fail.

4.47 Federal Information Processing Standards (FIPS) CryptoDev Validation

4.47.1 Overview

Federal Information Processing Standards (FIPS) are publicly announced standards developed by the United States federal government for use in computer systems by non-military government agencies and government contractors.

This application is used to parse and perform symmetric cryptography computation to the NIST Cryptographic Algorithm Validation Program (CAVP) test vectors.

For an algorithm implementation to be listed on a cryptographic module validation certificate as an Approved security function, the algorithm implementation must meet all the requirements of FIPS 140-2 and must successfully complete the cryptographic algorithm validation process.

4.47.2 Limitations

- Only NIST CAVP request files are parsed by this application.
- The version of request file supported is `CAVS 21.0`
- If the header comment in a `.req` file does not contain a Algo tag i.e `AES,TDES,GCM` you need to manually add it into the header comment for example:

```
# VARIABLE KEY - KAT for CBC / # TDES VARIABLE KEY - KAT for CBC
```

- The application does not supply the test vectors. The user is expected to obtain the test vector files from NIST website. To obtain the `.req` files you need to email a person from the NIST website and pay for the `.req` files. The `.rsp` files from the site can be used to validate and compare with the `.rsp` files created by the FIPS application.
- **Supported test vectors**
  - AES-CBC (128,192,256) - GFSbox, KeySbox, MCT, MMT
  - AES-GCM (128,192,256) - EncryptExtIV, Decrypt
  - AES-CCM (128) - VADT, VNT, VPT, VTT, DVPT
  - AES-CMAC (128) - Generate, Verify
  - HMAC (SHA1, SHA224, SHA256, SHA384, SHA512)
  - TDES-CBC (1 Key, 2 Keys, 3 Keys) - MMT, Monte, Permop, Subkey, Varkey, VarText

4.47.3 Application Information

If a `.req` is used as the input file after the application is finished running it will generate a response file or `.rsp`. Differences between the two files are, the `.req` file has missing information for instance if doing encryption you will not have the cipher text and that will be generated in the response file. Also if doing decryption it will not have the plain text until it finished the work and in the response file it will be added onto the end of each operation.

The application can be run with a `.rsp` file and what the outcome of that will be is it will add a extra line in the generated `.rsp` which should be the same as the `.rsp` used to run the application, this is useful for validating if the application has done the operation correctly.

4.47.4 Compiling the Application

- Compile Application

```
make -C examples/fips_validation
```

- Run `dos2unix` on the request files

```
dos2unix AES/req/*
dos2unix AES_GCM/req/*
dos2unix CCM/req/*
dos2unix CMAC/req/*
dos2unix HMAC/req/*
dos2unix TDES/req/*
```

4.47.5 Running the Application

The application requires a number of command line options:

```
./fips_validation [EAL options]
-- --req-file FILE_PATH/FOLDER_PATH
--rsp-file FILE_PATH/FOLDER_PATH
[--cryptodev DEVICE_NAME] [--cryptodev-id ID] [--path-is-folder]
```

where,

- req-file: The path of the request file or folder, separated by `path-is-folder` option.
- rsp-file: The path that the response file or folder is stored. separated by `path-is-folder` option.
- cryptodev: The name of the target DPDK Crypto device to be validated.
- cryptodev-id: The id of the target DPDK Crypto device to be validated.
- path-is-folder: If presented the application expects req-file and rsp-file are folder paths.

To run the application in linux environment to test one AES FIPS test data file for crypto_aesni_mb PMD, issue the command:

```
$ ./fips_validation --vdev crypto_aesni_mb --
--req-file /PATH/TO/REQUEST/FILE.req --rsp-file ./PATH/TO/RESPONSE/FILE.rsp
--cryptodev crypto_aesni_mb
```

To run the application in linux environment to test all AES-GCM FIPS test data files in one folder for crypto_aesni_gcm PMD, issue the command:

```
$ ./fips_validation --vdev crypto_aesni_gcm0 --
--req-file /PATH/TO/REQUEST/FILE/FOLDER/
--rsp-file ./PATH/TO/RESPONSE/FILE/FOLDER/
--cryptodev-id 0 --path-is-folder
```

4.48 IPsec Security Gateway Sample Application

The IPsec Security Gateway application is an example of a "real world" application using DPDK cryptodev framework.

4.48.1 Overview

The application demonstrates the implementation of a Security Gateway (not IPsec compliant, see the Constraints section below) using DPDK based on RFC4301, RFC4303, RFC3602 and RFC2404.

Internet Key Exchange (IKE) is not implemented, so only manual setting of Security Policies and Security Associations is supported.

The Security Policies (SP) are implemented as ACL rules, the Security Associations (SA) are stored in a table and the routing is implemented using LPM.

The application classifies the ports as *Protected* and *Unprotected*. Thus, traffic received on an Unprotected or Protected port is consider Inbound or Outbound respectively.

The application also supports complete IPsec protocol offload to hardware (Look aside crypto accelerator or using ethernet device). It also support inline ipsec processing by the supported ethernet device during transmission. These modes can be selected during the SA creation configuration.

In case of complete protocol offload, the processing of headers(ESP and outer IP header) is done by the hardware and the application does not need to add/remove them during outbound/inbound processing.

For inline offloaded outbound traffic, the application will not do the LPM lookup for routing, as the port on which the packet has to be forwarded will be part of the SA. Security parameters will be configured on that port only, and sending the packet on other ports could result in unencrypted packets being sent out.

The Path for IPsec Inbound traffic is:

- Read packets from the port.
- Classify packets between IPv4 and ESP.
- Perform Inbound SA lookup for ESP packets based on their SPI.
- Perform Verification/Decryption (Not needed in case of inline ipsec).
- Remove ESP and outer IP header (Not needed in case of protocol offload).
- Inbound SP check using ACL of decrypted packets and any other IPv4 packets.
- Routing.
- Write packet to port.

The Path for the IPsec Outbound traffic is:

- Read packets from the port.
- Perform Outbound SP check using ACL of all IPv4 traffic.
- Perform Outbound SA lookup for packets that need IPsec protection.
- Add ESP and outer IP header (Not needed in case protocol offload).
- Perform Encryption/Digest (Not needed in case of inline ipsec).

- Routing.
- Write packet to port.

The application supports two modes of operation: poll mode and event mode.

- In the poll mode a core receives packets from statically configured list of eth ports and eth ports' queues.
- In the event mode a core receives packets as events. After packet processing is done core submits them back as events to an event device. This enables multicore scaling and HW assisted scheduling by making use of the event device capabilities. The event mode configuration is predefined. All packets reaching given eth port will arrive at the same event queue. All event queues are mapped to all event ports. This allows all cores to receive traffic from all ports. Since the underlying event device might have varying capabilities, the worker threads can be drafted differently to maximize performance. For example, if an event device - eth device pair has Tx internal port, then application can call rte_event_eth_tx_adapter_enqueue() instead of regular rte_event_enqueue_burst(). So a thread which assumes that the device pair has internal port will not be the right solution for another pair. The infrastructure added for the event mode aims to help application to have multiple worker threads by maximizing performance from every type of event device without affecting existing paths/use cases. The worker to be used will be determined by the operating conditions and the underlying device capabilities. **Currently the application provides non-burst, internal port worker threads and supports inline protocol only.** It also provides infrastructure for non-internal port however does not define any worker threads.

Additionally the event mode introduces two submodes of processing packets:

- Driver submode: This submode has bare minimum changes in the application to support IPsec. There are no lookups, no routing done in the application. And for inline protocol use case, the worker thread resembles l2fwd worker thread as the IPsec processing is done entirely in HW. This mode can be used to benchmark the raw performance of the HW. The driver submode is selected with –single-sa option (used also by poll mode). When –single-sa option is used in conjution with event mode then index passed to –single-sa is ignored.
- App submode: This submode has all the features currently implemented with the application (non librte_ipsec path). All the lookups, routing follows existing methods and report numbers that can be compared against regular poll mode benchmark numbers.

4.48.2 Constraints

- No IPv6 options headers.
- No AH mode.
- Supported algorithms: AES-CBC, AES-CTR, AES-GCM, 3DES-CBC, HMAC-SHA1 and NULL.
- Each SA must be handle by a unique lcore (*1 RX queue per port*).

4.48.3 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ipsec-secgw` sub-directory.

1. [Optional] Build the application for debugging: This option adds some extra flags, disables compiler optimizations and is verbose:

```
make DEBUG=1
```

4.48.4 Running the Application

The application has a number of command line options:

```
./build/ipsec-secgw [EAL options] --
                     -p PORTMASK -P -u PORTMASK -j FRAMESIZE
                     -l -w REPLAY_WINOW_SIZE -e -a
                     -c SAD_CACHE_SIZE
                     -s NUMBER_OF_MBUFS_IN_PACKET_POOL
                     -f CONFIG_FILE_PATH
                     --config (port,queue,lcore)[,(port,queue,lcore)]
                     --single-sa SAIDX
                     --cryptodev_mask MASK
                     --transfer-mode MODE
                     --event-schedule-type TYPE
                     --rxoffload MASK
                     --txoffload MASK
                     --reassemble NUM
                     --mtu MTU
                     --frag-ttl FRAG_TTL_NS
```

Where:

- `-p PORTMASK`: Hexadecimal bitmask of ports to configure.
- `-P`: *optional*. Sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted (default is enabled).
- `-u PORTMASK`: hexadecimal bitmask of unprotected ports
- `-j FRAMESIZE`: *optional*. data buffer size (in bytes), in other words maximum data size for one segment. Packets with length bigger then FRAMESIZE still can be received, but will be segmented. Default value: RTE_MBUF_DEFAULT_BUF_SIZE (2176) Minimum value: RTE_MBUF_DEFAULT_BUF_SIZE (2176) Maximum value: UINT16_MAX (65535).
- `-l`: enables code-path that uses librte_ipsec.
- `-w REPLAY_WINOW_SIZE`: specifies the IPsec sequence number replay window size for each Security Association (available only with librte_ipsec code path).
- `-e`: enables Security Association extended sequence number processing (available only with librte_ipsec code path).
- `-a`: enables Security Association sequence number atomic behavior (available only with librte_ipsec code path).

- `-c`: specifies the SAD cache size. Stores the most recent SA in a per lcore cache. Cache represents flat array containing SA's indexed by SPI. Zero value disables cache. Default value: 128.
- `-s`: sets number of mbufs in packet pool, if not provided number of mbufs will be calculated based on number of cores, eth ports and crypto queues.
- `-f CONFIG_FILE_PATH`: the full path of text-based file containing all configuration items for running the application (See Configuration file syntax section below). `-f CONFIG_FILE_PATH` **must** be specified. **ONLY** the UNIX format configuration file is accepted.
- `--config (port,queue,lcore)[,(port,queue,lcore)]`: in poll mode determines which queues from which ports are mapped to which cores. In event mode this option is not used as packets are dynamically scheduled to cores by HW.
- `--single-sa SAIDX`: in poll mode use a single SA for outbound traffic, bypassing the SP on both Inbound and Outbound. This option is meant for debugging/performance purposes. In event mode selects driver submode, SA index value is ignored.
- `--cryptodev_mask MASK`: hexadecimal bitmask of the crypto devices to configure.
- `--transfer-mode MODE`: sets operating mode of the application "poll" : packet transfer via polling (default) "event" : Packet transfer via event device
- `--event-schedule-type TYPE`: queue schedule type, applies only when –transfer-mode is set to event. "ordered" : Ordered (default) "atomic" : Atomic "parallel" : Parallel When –event-schedule-type is set as RTE_SCHED_TYPE_ORDERED/ATOMIC, event device will ensure the ordering. Ordering will be lost when tried in PARALLEL.
- `--rxoffload MASK`: RX HW offload capabilities to enable/use on this port (bitmask of DEV_RX_OFFLOAD_* values). It is an optional parameter and allows user to disable some of the RX HW offload capabilities. By default all HW RX offloads are enabled.
- `--txoffload MASK`: TX HW offload capabilities to enable/use on this port (bitmask of DEV_TX_OFFLOAD_* values). It is an optional parameter and allows user to disable some of the TX HW offload capabilities. By default all HW TX offloads are enabled.
- `--reassemble NUM`: max number of entries in reassemble fragment table. Zero value disables reassembly functionality. Default value: 0.
- `--mtu MTU`: MTU value (in bytes) on all attached ethernet ports. Outgoing packets with length bigger then MTU will be fragmented. Incoming packets with length bigger then MTU will be discarded. Default value: 1500.
- `--frag-ttl FRAG_TTL_NS`: fragment lifetime (in nanoseconds). If packet is not reassembled within this time, received fragments will be discarded. Fragment lifetime should be decreased when there is a high fragmented traffic loss in high bandwidth networks. Should be lower for low number of reassembly buckets. Valid values: from 1 ns to 10 s. Default value: 10000000 (10 s).

The mapping of lcores to port/queues is similar to other l3fwd applications.

For example, given the following command line to run application in poll mode:

```
./build/ipsec-secgw -l 20,21 -n 4 --socket-mem 0,2048       \
      --vdev "crypto_null" -- -p 0xf -P -u 0x3              \
      --config="(0,0,20),(1,0,20),(2,0,21),(3,0,21)"        \
      -f /path/to/config_file --transfer-mode poll          \
```

where each option means:

- The `-l` option enables cores 20 and 21.

- The `-n` option sets memory 4 channels.
- The `--socket-mem` to use 2GB on socket 1.
- The `--vdev "crypto_null"` option creates virtual NULL cryptodev PMD.
- The `-p` option enables ports (detected) 0, 1, 2 and 3.
- The `-P` option enables promiscuous mode.
- The `-u` option sets ports 0 and 1 as unprotected, leaving 2 and 3 as protected.
- The `--config` option enables one queue per port with the following mapping:

Port	Queue	lcore	Description
0	0	20	Map queue 0 from port 0 to lcore 20.
1	0	20	Map queue 0 from port 1 to lcore 20.
2	0	21	Map queue 0 from port 2 to lcore 21.
3	0	21	Map queue 0 from port 3 to lcore 21.

- The `-f /path/to/config_file` option enables the application read and parse the configuration file specified, and configures the application with a given set of SP, SA and Routing entries accordingly. The syntax of the configuration file will be explained below in more detail. Please **note** the parser only accepts UNIX format text file. Other formats such as DOS/MAC format will cause a parse error.
- The `--transfer-mode` option selects poll mode for processing packets.

Similarly for example, given the following command line to run application in event app mode:

```
./build/ipsec-secgw -c 0x3 -- -P -p 0x3 -u 0x1       \
       -f /path/to/config_file --transfer-mode event \
       --event-schedule-type parallel                \
```

where each option means:

- The `-c` option selects cores 0 and 1 to run on.
- The `-P` option enables promiscuous mode.
- The `-p` option enables ports (detected) 0 and 1.
- The `-u` option sets ports 0 as unprotected, leaving 1 as protected.
- The `-f /path/to/config_file` option has the same behavior as in poll mode example.
- The `--transfer-mode` option selects event mode for processing packets.
- The `--event-schedule-type` option selects parallel ordering of event queues.

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The application would do a best effort to "map" crypto devices to cores, with hardware devices having priority. Basically, hardware devices if present would be assigned to a core before software ones. This means that if the application is using a single core and both hardware and software crypto devices are detected, hardware devices will be used.

A way to achieve the case where you want to force the use of virtual crypto devices is to whitelist the Ethernet devices needed and therefore implicitly blacklisting all hardware crypto devices.

For example, something like the following command line:

```
./build/ipsec-secgw -l 20,21 -n 4 --socket-mem 0,2048 \
        -w 81:00.0 -w 81:00.1 -w 81:00.2 -w 81:00.3 \
        --vdev "crypto_aesni_mb" --vdev "crypto_null" \
        -- \
        -p 0xf -P -u 0x3 --config="(0,0,20),(1,0,20),(2,0,21),(3,0,21)" \
        -f sample.cfg
```

4.48.5 Configurations

The following sections provide the syntax of configurations to initialize your SP, SA, Routing and Neighbour tables. Configurations shall be specified in the configuration file to be passed to the application. The file is then parsed by the application. The successful parsing will result in the appropriate rules being applied to the tables accordingly.

Configuration File Syntax

As mention in the overview, the Security Policies are ACL rules. The application parsers the rules specified in the configuration file and passes them to the ACL table, and replicates them per socket in use.

Following are the configuration file syntax.

General rule syntax

The parse treats one line in the configuration file as one configuration item (unless the line concatenation symbol exists). Every configuration item shall follow the syntax of either SP, SA, Routing or Neighbour rules specified below.

The configuration parser supports the following special symbols:

- Comment symbol #. Any character from this symbol to the end of line is treated as comment and will not be parsed.
- Line concatenation symbol \. This symbol shall be placed in the end of the line to be concatenated to the line below. Multiple lines' concatenation is supported.

SP rule syntax

The SP rule syntax is shown as follows:

```
sp <ip_ver> <dir> esp <action> <priority> <src_ip> <dst_ip>
<proto> <sport> <dport>
```

where each options means:

<ip_ver>

- IP protocol version
- Optional: No
- Available options:

  - *ipv4*: IP protocol version 4
  - *ipv6*: IP protocol version 6

<dir>

- The traffic direction
- Optional: No
- Available options:
  - *in*: inbound traffic
  - *out*: outbound traffic

<action>

- IPsec action
- Optional: No
- Available options:
  - *protect* <*SA_idx*>: the specified traffic is protected by SA rule with id SA_idx
  - *bypass*: the specified traffic traffic is bypassed
  - *discard*: the specified traffic is discarded

<priority>

- Rule priority
- Optional: Yes, default priority 0 will be used
- Syntax: *pri* <*id*>

<src_ip>

- The source IP address and mask
- Optional: Yes, default address 0.0.0.0 and mask of 0 will be used
- Syntax:
  - *src X.X.X.X/Y* for IPv4
  - *src XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX/Y* for IPv6

<dst_ip>

- The destination IP address and mask
- Optional: Yes, default address 0.0.0.0 and mask of 0 will be used
- Syntax:
  - *dst X.X.X.X/Y* for IPv4
  - *dst XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX/Y* for IPv6

<proto>

- The protocol start and end range
- Optional: yes, default range of 0 to 0 will be used

- Syntax: *proto X:Y*

<sport>

- The source port start and end range
- Optional: yes, default range of 0 to 0 will be used
- Syntax: *sport X:Y*

<dport>

- The destination port start and end range
- Optional: yes, default range of 0 to 0 will be used
- Syntax: *dport X:Y*

Example SP rules:

```
sp ipv4 out esp protect 105 pri 1 dst 192.168.115.0/24 sport 0:65535 \
dport 0:65535

sp ipv6 in esp bypass pri 1 dst 0000:0000:0000:0000:5555:5555:\
0000:0000/96 sport 0:65535 dport 0:65535
```

SA rule syntax

The successfully parsed SA rules will be stored in an array table.

The SA rule syntax is shown as follows:

```
sa <dir> <spi> <cipher_algo> <cipher_key> <auth_algo> <auth_key>
<mode> <src_ip> <dst_ip> <action_type> <port_id> <fallback>
<flow-direction> <port_id> <queue_id>
```

where each options means:

<dir>

- The traffic direction
- Optional: No
- Available options:
  - *in*: inbound traffic
  - *out*: outbound traffic

<spi>

- The SPI number
- Optional: No
- Syntax: unsigned integer number

<cipher_algo>

- Cipher algorithm
- Optional: Yes, unless <aead_algo> is not used

- Available options:
  - *null*: NULL algorithm
  - *aes-128-cbc*: AES-CBC 128-bit algorithm
  - *aes-192-cbc*: AES-CBC 192-bit algorithm
  - *aes-256-cbc*: AES-CBC 256-bit algorithm
  - *aes-128-ctr*: AES-CTR 128-bit algorithm
  - *3des-cbc*: 3DES-CBC 192-bit algorithm
- Syntax: *cipher_algo <your algorithm>*

<cipher_key>

- Cipher key, NOT available when 'null' algorithm is used
- Optional: Yes, unless <aead_algo> is not used. Must be followed by <cipher_algo> option
- Syntax: Hexadecimal bytes (0x0-0xFF) concatenate by colon symbol ':'. The number of bytes should be as same as the specified cipher algorithm key size.

  For example: *cipher_key A1:B2:C3:D4:A1:B2:C3:D4:A1:B2:C3:D4: A1:B2:C3:D4*

<auth_algo>

- Authentication algorithm
- Optional: Yes, unless <aead_algo> is not used
- Available options:
  - *null*: NULL algorithm
  - *sha1-hmac*: HMAC SHA1 algorithm

<auth_key>

- Authentication key, NOT available when 'null' or 'aes-128-gcm' algorithm is used.
- Optional: Yes, unless <aead_algo> is not used. Must be followed by <auth_algo> option
- Syntax: Hexadecimal bytes (0x0-0xFF) concatenate by colon symbol ':'. The number of bytes should be as same as the specified authentication algorithm key size.

  For example: *auth_key A1:B2:C3:D4:A1:B2:C3:D4:A1:B2:C3:D4:A1:B2:C3:D4: A1:B2:C3:D4*

<aead_algo>

- AEAD algorithm
- Optional: Yes, unless <cipher_algo> and <auth_algo> are not used
- Available options:
  - *aes-128-gcm*: AES-GCM 128-bit algorithm
  - *aes-192-gcm*: AES-GCM 192-bit algorithm
  - *aes-256-gcm*: AES-GCM 256-bit algorithm
- Syntax: *cipher_algo <your algorithm>*

<aead_key>

- Cipher key, NOT available when 'null' algorithm is used
- Optional: Yes, unless <cipher_algo> and <auth_algo> are not used. Must be followed by <aead_algo> option
- Syntax: Hexadecimal bytes (0x0-0xFF) concatenate by colon symbol ':'. Last 4 bytes of the provided key will be used as 'salt' and so, the number of bytes should be same as the sum of specified AEAD algorithm key size and salt size (4 bytes).

  For example: *aead_key A1:B2:C3:D4:A1:B2:C3:D4:A1:B2:C3:D4:A1:B2:C3:D4:A1:B2:C3:D4*

<mode>

- The operation mode
- Optional: No
- Available options:
  - *ipv4-tunnel*: Tunnel mode for IPv4 packets
  - *ipv6-tunnel*: Tunnel mode for IPv6 packets
  - *transport*: transport mode
- Syntax: mode XXX

<src_ip>

- The source IP address. This option is not available when transport mode is used
- Optional: Yes, default address 0.0.0.0 will be used
- Syntax:
  - *src X.X.X.X* for IPv4
  - *src XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX* for IPv6

<dst_ip>

- The destination IP address. This option is not available when transport mode is used
- Optional: Yes, default address 0.0.0.0 will be used
- Syntax:
  - *dst X.X.X.X* for IPv4
  - *dst XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX* for IPv6

<type>

- Action type to specify the security action. This option specify the SA to be performed with look aside protocol offload to HW accelerator or protocol offload on ethernet device or inline crypto processing on the ethernet device during transmission.
- Optional: Yes, default type *no-offload*
- Available options:
  - *lookaside-protocol-offload*: look aside protocol offload to HW accelerator

  - *inline-protocol-offload*: inline protocol offload on ethernet device
  - *inline-crypto-offload*: inline crypto processing on ethernet device
  - *no-offload*: no offloading to hardware

<port_id>

- Port/device ID of the ethernet/crypto accelerator for which the SA is configured. For *inline-crypto-offload* and *inline-protocol-offload*, this port will be used for routing. The routing table will not be referred in this case.
- Optional: No, if *type* is not *no-offload*
- Syntax:
  - *port_id X* X is a valid device number in decimal

<fallback>

- Action type for ingress IPsec packets that inline processor failed to process. Only a combination of *inline-crypto-offload* as a primary session and *lookaside-none* as a fall-back session is supported at the moment.

  If used in conjunction with IPsec window, its width needs be increased due to different processing times of inline and lookaside modes which results in packet reordering.
- Optional: Yes.
- Available options:
  - *lookaside-none*: use automatically chosen cryptodev to process packets
- Syntax:
  - *fallback lookaside-none*

<flow-direction>

- Option for redirecting a specific inbound ipsec flow of a port to a specific queue of that port.
- Optional: Yes.
- Available options:
  - *port_id*: Port ID of the NIC for which the SA is configured.
  - *queue_id*: Queue ID to which traffic should be redirected.

Example SA rules:

```
sa out 5 cipher_algo null auth_algo null mode ipv4-tunnel \
src 172.16.1.5 dst 172.16.2.5

sa out 25 cipher_algo aes-128-cbc \
cipher_key c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3 \
auth_algo sha1-hmac \
auth_key c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3:c3 \
mode ipv6-tunnel \
src 1111:1111:1111:1111:1111:1111:1111:5555 \
dst 2222:2222:2222:2222:2222:2222:2222:5555

sa in 105 aead_algo aes-128-gcm \
aead_key de:ad:be:ef:de:ad:be:ef:de:ad:be:ef:de:ad:be:ef \
```

(continues on next page)

(continued from previous page)

```
mode ipv4-tunnel src 172.16.2.5 dst 172.16.1.5

sa out 5 cipher_algo aes-128-cbc cipher_key 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0 \
auth_algo sha1-hmac auth_key 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0 \
mode ipv4-tunnel src 172.16.1.5 dst 172.16.2.5 \
type lookaside-protocol-offload port_id 4

sa in 35 aead_algo aes-128-gcm \
aead_key de:ad:be:ef:de:ad:be:ef:de:ad:be:ef:de:ad:be:ef \
mode ipv4-tunnel src 172.16.2.5 dst 172.16.1.5 \
type inline-crypto-offload port_id 0

sa in 117 cipher_algo null auth_algo null mode ipv4-tunnel src 172.16.2.7 \
dst 172.16.1.7 flow-direction 0 2
```

Routing rule syntax

The Routing rule syntax is shown as follows:

```
rt <ip_ver> <src_ip> <dst_ip> <port>
```

where each options means:

<ip_ver>

- IP protocol version
- Optional: No
- Available options:
  - *ipv4*: IP protocol version 4
  - *ipv6*: IP protocol version 6

<src_ip>

- The source IP address and mask
- Optional: Yes, default address 0.0.0.0 and mask of 0 will be used
- Syntax:
  - *src X.X.X.X/Y* for IPv4
  - *src XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX/Y* for IPv6

<dst_ip>

- The destination IP address and mask
- Optional: Yes, default address 0.0.0.0 and mask of 0 will be used
- Syntax:
  - *dst X.X.X.X/Y* for IPv4
  - *dst XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX/Y* for IPv6

<port>

- The traffic output port id

- Optional: yes, default output port 0 will be used
- Syntax: *port X*

Example SP rules:

```
rt ipv4 dst 172.16.1.5/32 port 0

rt ipv6 dst 1111:1111:1111:1111:1111:1111:1111:5555/116 port 0
```

Neighbour rule syntax

The Neighbour rule syntax is shown as follows:

```
neigh <port> <dst_mac>
```

where each options means:

<port>

- The output port id
- Optional: No
- Syntax: *port X*

<dst_mac>

- The destination ethernet address to use for that port
- Optional: No
- Syntax:
  - XX:XX:XX:XX:XX:XX

Example Neighbour rules:

```
neigh port 0 DE:AD:BE:EF:01:02
```

4.48.6 Test directory

The test directory contains scripts for testing the various encryption algorithms.

The purpose of the scripts is to automate ipsec-secgw testing using another system running linux as a DUT.

The user must setup the following environment variables:

- SGW_PATH: path to the ipsec-secgw binary to test.
- REMOTE_HOST: IP address/hostname of the DUT.
- REMOTE_IFACE: interface name for the test-port on the DUT.
- ETH_DEV: ethernet device to be used on the SUT by DPDK ('-w <pci-id>')

Also the user can optionally setup:

- SGW_LCORE: lcore to run ipsec-secgw on (default value is 0)

- `CRYPTO_DEV`: crypto device to be used ('-w <pci-id>'). If none specified appropriate vdevs will be created by the script

Scripts can be used for multiple test scenarios. To check all available options run:

```
/bin/bash run_test.sh -h
```

Note that most of the tests require the appropriate crypto PMD/device to be available.

Server configuration

Two servers are required for the tests, SUT and DUT.

Make sure the user from the SUT can ssh to the DUT without entering the password. To enable this feature keys must be setup on the DUT.

`ssh-keygen` will make a private & public key pair on the SUT.

`ssh-copy-id` <user name>@<target host name> on the SUT will copy the public key to the DUT. It will ask for credentials so that it can upload the public key.

The SUT and DUT are connected through at least 2 NIC ports.

One NIC port is expected to be managed by linux on both machines and will be used as a control path.

The second NIC port (test-port) should be bound to DPDK on the SUT, and should be managed by linux on the DUT.

The script starts `ipsec-secgw` with 2 NIC devices: `test-port` and `tap vdev`.

It then configures the local tap interface and the remote interface and IPsec policies in the following way:

Traffic going over the test-port in both directions has to be protected by IPsec.

Traffic going over the TAP port in both directions does not have to be protected.

i.e:

DUT OS(NIC1)–(IPsec)–>(NIC1)ipsec-secgw(TAP)–(plain)–>(TAP)SUT OS

SUT OS(TAP)–(plain)–>(TAP)psec-secgw(NIC1)–(IPsec)–>(NIC1)DUT OS

It then tries to perform some data transfer using the scheme described above.

Usage

In the ipsec-secgw/test directory run

/bin/bash run_test.sh <options> <ipsec_mode>

Available options:

- `-4` Perform tests with use of IPv4. One or both [-46] options needs to be selected.
- `-6` Perform tests with use of IPv6. One or both [-46] options needs to be selected.
- `-m` Add IPSec tunnel mixed IP version tests - outer IP version different than inner. Inner IP version will match selected option [-46].
- `-i` Run tests in inline mode. Regular tests will not be invoked.

- `-f` Run tests for fallback mechanism. Regular tests will not be invoked.
- `-l` Run tests in legacy mode only. It cannot be used with options [-fsc]. On default library mode is used.
- `-s` Run all tests with reassembly support. On default only tests for fallback mechanism use reassembly support.
- `-c` Run tests with use of cpu-crypto. For inline tests it will not be applied. On default lookaside-none is used.
- `-p` Perform packet validation tests. Option [-46] is not required.
- `-h` Show usage.

If <ipsec_mode> is specified, only tests for that mode will be invoked. For the list of available modes please refer to run_test.sh.

4.49 Loop-back Sample Application using Baseband Device (bbdev)

The baseband sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) for baseband workloads using Wireless Device abstraction library.

4.49.1 Overview

The Baseband device sample application performs a loop-back operation using a baseband device capable of transceiving data packets. A packet is received on an ethernet port -> enqueued for downlink baseband operation -> dequeued from the downlink baseband device -> enqueued for uplink baseband operation -> dequeued from the baseband device -> then the received packet is compared with the baseband operations output. Then it's looped back to the ethernet port.

- The MAC header is preserved in the packet

4.49.2 Limitations

- Only one baseband device and one ethernet port can be used.

4.49.3 Compiling the Application

1. DPDK needs to be built with `baseband_turbo_sw` PMD driver enabled along with `FLEXRAN SDK` Libraries. Refer to *SW Turbo Poll Mode Driver* documentation for more details on this.
2. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/bbdev_app
```

3. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linux-gcc
```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

4. Build the application:

```
make
```

4.49.4 Running the Application

The application accepts a number of command line options:

```
$ ./build/bbdev [EAL options] -- [-e ENCODING_CORES] [-d DECODING_CORES] /
[-p ETH_PORT_ID] [-b BBDEV_ID]
```

where:

- e ENCODING_CORES: hexmask for encoding lcores (default = 0x2)
- d DECODING_CORES: hexmask for decoding lcores (default = 0x4)
- p ETH_PORT_ID: ethernet port ID (default = 0)
- b BBDEV_ID: BBDev ID (default = 0)

The application requires that baseband devices is capable of performing the specified baseband operation are available on application initialization. This means that HW baseband device/s must be bound to a DPDK driver or a SW baseband device/s (virtual BBdev) must be created (using –vdev).

To run the application in linux environment with the turbo_sw baseband device using the whitelisted port running on 1 encoding lcore and 1 decoding lcore issue the command:

```
$ ./build/bbdev --vdev='baseband_turbo_sw' -w <NIC0PCIADDR> -c 0x38 --socket-mem=2,2 \
--file-prefix=bbdev -- -e 0x10 -d 0x20
```

where, NIC0PCIADDR is the PCI address of the Rx port

This command creates one virtual bbdev devices `baseband_turbo_sw` where the device gets linked to a corresponding ethernet port as whitelisted by the parameter -w. 3 cores are allocated to the application, and assigned as:

- core 3 is the master and used to print the stats live on screen,
- core 4 is the encoding lcore performing Rx and Turbo Encode operations
- core 5 is the downlink lcore performing Turbo Decode, validation and Tx operations

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.49.5 Using Packet Generator with baseband device sample application

To allow the bbdev sample app to do the loopback, an influx of traffic is required. This can be done by using DPDK Pktgen to burst traffic on two ethernet ports, and it will print the transmitted along with the looped-back traffic on Rx ports. Executing the command below will generate traffic on the two whitelisted ethernet ports.

```
$ ./pktgen-3.4.0/app/x86_64-native-linux-gcc/pktgen -c 0x3 \
--socket-mem=1,1 --file-prefix=pg -w <NIC1PCIADDR> -- -m 1.0 -P
```

where:

- `-c COREMASK`: A hexadecimal bitmask of cores to run on
- `--socket-mem`: Memory to allocate on specific sockets (use comma separated values)
- `--file-prefix`: Prefix for hugepage filenames
- `-w <NIC1PCIADDR>`: Add a PCI device in white list. The argument format is <[domain:]bus:devid.func>.
- `-m <string>`: Matrix for mapping ports to logical cores.
- `-P`: PROMISCUOUS mode

Refer to *The Pktgen Application* documents for general information on running Pktgen with DPDK applications.

4.50 NTB Sample Application

The ntb sample application shows how to use ntb rawdev driver. This sample provides interactive mode to do packet based processing between two systems.

This sample supports 4 types of packet forwarding mode.

- `file-trans`: transmit files between two systems. The sample will be polling to receive files from the peer and save the file as `ntb_recv_file[N]`, [N] represents the number of received file.
- `rxonly`: NTB receives packets but doesn't transmit them.
- `txonly`: NTB generates and transmits packets without receiving any.
- `iofwd`: iofwd between NTB device and ethdev.

4.50.1 Compiling the Application

To compile the sample application see *Compiling the Sample Applications*.

The application is located in the `ntb` sub-directory.

4.50.2 Running the Application

The application requires an available core for each port, plus one. The only available options are the standard ones for the EAL:

```
./build/ntb_fwd -c 0xf -n 6 -- -i
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

4.50.3 Command-line Options

The application supports the following command-line options.

- `--buf-size=N`

  Set the data size of the mbufs used to N bytes, where N < 65536. The default value is 2048.

- `--fwd-mode=mode`

  Set the packet forwarding mode as `file-trans`, `txonly`, `rxonly` or `iofwd`.

- `--nb-desc=N`

  Set number of descriptors of queue as N, namely queue size, where 64 <= N <= 1024. The default value is 1024.

- `--txfreet=N`

  Set the transmit free threshold of TX rings to N, where 0 <= N <= the value of `--nb-desc`. The default value is 256.

- `--burst=N`

  Set the number of packets per burst to N, where 1 <= N <= 32. The default value is 32.

- `--qp=N`

  Set the number of queues as N, where qp > 0. The default value is 1.

4.50.4 Using the application

The application is console-driven using the cmdline DPDK interface:

```
ntb>
```

From this interface the available commands and descriptions of what they do as follows:

- `send [filepath]`: Send file to the peer host. Need to be in file-trans forwarding mode first.
- `start`: Start transmission.
- `stop`: Stop transmission.
- `show/clear port stats`: Show/Clear port stats and throughput.
- `set fwd file-trans/rxonly/txonly/iofwd`: Set packet forwarding mode.
- `quit`: Exit program.

CHAPTER

FIVE

PROGRAMMER'S GUIDE

5.1 Introduction

This document provides software architecture information, development environment information and optimization guidelines.

For programming examples and for instructions on compiling and running each sample application, see the *DPDK Sample Applications User Guide* for details.

For general information on compiling and running applications, see the *DPDK Getting Started Guide*.

5.1.1 Documentation Roadmap

The following is a list of DPDK documents in the suggested reading order:

- **Release Notes** : Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- **Getting Started Guide** : Describes how to install and configure the DPDK software; designed to get users up and running quickly with the software.
- **FreeBSD* Getting Started Guide** : A document describing the use of the DPDK with FreeBSD* has been added in DPDK Release 1.6.0. Refer to this guide for installation and configuration instructions to get started using the DPDK with FreeBSD*.
- **Programmer's Guide** (this document): Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux* application (linux) environment
  - The content of the DPDK, the build system (including the commands that can be used in the root DPDK Makefile to build the development kit and an application) and guidelines for porting an application
  - Optimizations used in the software and those that should be considered for new development

  A glossary of terms is also provided.
- **API Reference** : Provides detailed information about DPDK functions, data structures and other programming constructs.
- **Sample Applications User Guide**: Describes a set of sample applications. Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

5.1.2 Related Publications

The following documents provide information that is relevant to the development of applications using the DPDK:

- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide

Part 1: Architecture Overview

5.2 Overview

This section gives a global overview of the architecture of Data Plane Development Kit (DPDK).

The main goal of the DPDK is to provide a simple, complete framework for fast packet processing in data plane applications. Users may use the code to understand some of the techniques employed, to build upon for prototyping or to add their own protocol stacks. Alternative ecosystem options that use the DPDK are available.

The framework creates a set of libraries for specific environments through the creation of an Environment Abstraction Layer (EAL), which may be specific to a mode of the Intel® architecture (32-bit or 64-bit), Linux* user space compilers or a specific platform. These environments are created through the use of make files and configuration files. Once the EAL library is created, the user may link with the library to create their own applications. Other libraries, outside of EAL, including the Hash, Longest Prefix Match (LPM) and rings libraries are also provided. Sample applications are provided to help show the user how to use various features of the DPDK.

The DPDK implements a run to completion model for packet processing, where all resources must be allocated prior to calling Data Plane applications, running as execution units on logical processing cores. The model does not support a scheduler and all devices are accessed by polling. The primary reason for not using interrupts is the performance overhead imposed by interrupt processing.

In addition to the run-to-completion model, a pipeline model may also be used by passing packets or messages between cores via the rings. This allows work to be performed in stages and may allow more efficient use of code on cores.

5.2.1 Development Environment

The DPDK project installation requires Linux and the associated toolchain, such as one or more compilers, assembler, make utility, editor and various libraries to create the DPDK components and libraries.

Once these libraries are created for the specific environment and architecture, they may then be used to create the user's data plane application.

When creating applications for the Linux user space, the glibc library is used. For DPDK applications, two environmental variables (RTE_SDK and RTE_TARGET) must be configured before compiling the applications. The following are examples of how the variables can be set:

```
export RTE_SDK=/home/user/DPDK
export RTE_TARGET=x86_64-native-linux-gcc
```

See the *DPDK Getting Started Guide* for information on setting up the development environment.

5.2.2 Environment Abstraction Layer

The Environment Abstraction Layer (EAL) provides a generic interface that hides the environment specifics from the applications and libraries. The services provided by the EAL are:

- DPDK loading and launching
- Support for multi-process and multi-thread execution types
- Core affinity/assignment procedures
- System memory allocation/de-allocation
- Atomic/lock operations
- Time reference
- PCI bus access
- Trace and debug functions
- CPU feature identification
- Interrupt handling
- Alarm operations
- Memory management (malloc)

The EAL is fully described in *Environment Abstraction Layer*.

5.2.3 Core Components

The *core components* are a set of libraries that provide all the elements needed for high-performance packet processing applications.

Fig. 5.1: Core Components Architecture

Ring Manager (librte_ring)

The ring structure provides a lockless multi-producer, multi-consumer FIFO API in a finite size table. It has some advantages over lockless queues; easier to implement, adapted to bulk operations and faster. A ring is used by the *Memory Pool Manager (librte_mempool)* and may be used as a general communication mechanism between cores and/or execution blocks connected together on a logical core.

This ring buffer and its usage are fully described in *Ring Library*.

Memory Pool Manager (librte_mempool)

The Memory Pool Manager is responsible for allocating pools of objects in memory. A pool is identified by name and uses a ring to store free objects. It provides some other optional services, such as a per-core object cache and an alignment helper to ensure that objects are padded to spread them equally on all RAM channels.

This memory pool allocator is described in *Mempool Library*.

Network Packet Buffer Management (librte_mbuf)

The mbuf library provides the facility to create and destroy buffers that may be used by the DPDK application to store message buffers. The message buffers are created at startup time and stored in a mempool, using the DPDK mempool library.

This library provides an API to allocate/free mbufs, manipulate packet buffers which are used to carry network packets.

Network Packet Buffer Management is described in *Mbuf Library*.

Timer Manager (librte_timer)

This library provides a timer service to DPDK execution units, providing the ability to execute a function asynchronously. It can be periodic function calls, or just a one-shot call. It uses the timer interface provided by the Environment Abstraction Layer (EAL) to get a precise time reference and can be initiated on a per-core basis as required.

The library documentation is available in *Timer Library*.

5.2.4 Ethernet* Poll Mode Driver Architecture

The DPDK includes Poll Mode Drivers (PMDs) for 1 GbE, 10 GbE and 40GbE, and para virtualized virtio Ethernet controllers which are designed to work without asynchronous, interrupt-based signaling mechanisms.

See *Poll Mode Driver*.

5.2.5 Packet Forwarding Algorithm Support

The DPDK includes Hash (librte_hash) and Longest Prefix Match (LPM,librte_lpm) libraries to support the corresponding packet forwarding algorithms.

See *Hash Library* and *LPM Library* for more information.

5.2.6 librte_net

The librte_net library is a collection of IP protocol definitions and convenience macros. It is based on code from the FreeBSD* IP stack and contains protocol numbers (for use in IP headers), IP-related macros, IPv4/IPv6 header structures and TCP, UDP and SCTP header structures.

5.3 Environment Abstraction Layer

The Environment Abstraction Layer (EAL) is responsible for gaining access to low-level resources such as hardware and memory space. It provides a generic interface that hides the environment specifics from the applications and libraries. It is the responsibility of the initialization routine to decide how to allocate these resources (that is, memory space, devices, timers, consoles, and so on).

Typical services expected from the EAL are:

- DPDK Loading and Launching: The DPDK and its application are linked as a single application and must be loaded by some means.
- Core Affinity/Assignment Procedures: The EAL provides mechanisms for assigning execution units to specific cores as well as creating execution instances.
- System Memory Reservation: The EAL facilitates the reservation of different memory zones, for example, physical memory areas for device interactions.
- Trace and Debug Functions: Logs, dump_stack, panic and so on.
- Utility Functions: Spinlocks and atomic counters that are not provided in libc.
- CPU Feature Identification: Determine at runtime if a particular feature, for example, Intel® AVX is supported. Determine if the current CPU supports the feature set that the binary was compiled for.
- Interrupt Handling: Interfaces to register/unregister callbacks to specific interrupt sources.
- Alarm Functions: Interfaces to set/remove callbacks to be run at a specific time.

5.3.1 EAL in a Linux-userland Execution Environment

In a Linux user space environment, the DPDK application runs as a user-space application using the pthread library.

The EAL performs physical memory allocation using mmap() in hugetlbfs (using huge page sizes to increase performance). This memory is exposed to DPDK service layers such as the *Mempool Library*.

At this point, the DPDK services layer will be initialized, then through pthread setaffinity calls, each execution unit will be assigned to a specific logical core to run as a user-level thread.

The time reference is provided by the CPU Time-Stamp Counter (TSC) or by the HPET kernel API through a mmap() call.

Initialization and Core Launching

Part of the initialization is done by the start function of glibc. A check is also performed at initialization time to ensure that the micro architecture type chosen in the config file is supported by the CPU. Then, the main() function is called. The core initialization and launch is done in rte_eal_init() (see the API documentation). It consist of calls to the pthread library (more specifically, pthread_self(), pthread_create(), and pthread_setaffinity_np()).

Fig. 5.2: EAL Initialization in a Linux Application Environment

Note: Initialization of objects, such as memory zones, rings, memory pools, lpm tables and hash tables, should be done as part of the overall application initialization on the master lcore. The creation and initialization functions for these objects are not multi-thread safe. However, once initialized, the objects themselves can safely be used in multiple threads simultaneously.

Shutdown and Cleanup

During the initialization of EAL resources such as hugepage backed memory can be allocated by core components. The memory allocated during `rte_eal_init()` can be released by calling the `rte_eal_cleanup()` function. Refer to the API documentation for details.

Multi-process Support

The Linux EAL allows a multi-process as well as a multi-threaded (pthread) deployment model. See chapter *Multi-process Support* for more details.

Memory Mapping Discovery and Memory Reservation

The allocation of large contiguous physical memory is done using the hugetlbfs kernel filesystem. The EAL provides an API to reserve named memory zones in this contiguous memory. The physical address of the reserved memory for that memory zone is also returned to the user by the memory zone reservation API.

There are two modes in which DPDK memory subsystem can operate: dynamic mode, and legacy mode. Both modes are explained below.

Note: Memory reservations done using the APIs provided by rte_malloc are also backed by pages from the hugetlbfs filesystem.

- Dynamic memory mode

Currently, this mode is only supported on Linux.

In this mode, usage of hugepages by DPDK application will grow and shrink based on application's requests. Any memory allocation through `rte_malloc()`, `rte_memzone_reserve()` or other methods, can potentially result in more hugepages being reserved from the system. Similarly, any memory deallocation can potentially result in hugepages being released back to the system.

Memory allocated in this mode is not guaranteed to be IOVA-contiguous. If large chunks of IOVA-contiguous are required (with "large" defined as "more than one page"), it is recommended to either use VFIO driver for all physical devices (so that IOVA and VA addresses can be the same, thereby bypassing physical addresses entirely), or use legacy memory mode.

For chunks of memory which must be IOVA-contiguous, it is recommended to use `rte_memzone_reserve()` function with `RTE_MEMZONE_IOVA_CONTIG` flag specified. This way, memory allocator will ensure that, whatever memory mode is in use, either reserved memory will satisfy the requirements, or the allocation will fail.

There is no need to preallocate any memory at startup using `-m` or `--socket-mem` command-line parameters, however it is still possible to do so, in which case preallocate memory will be "pinned" (i.e. will never be released by the application back to the system). It will be possible to allocate more hugepages, and deallocate those, but any preallocated pages will not be freed. If neither `-m` nor `--socket-mem` were specified, no memory will be preallocated, and all memory will be allocated at runtime, as needed.

Another available option to use in dynamic memory mode is `--single-file-segments` command-line option. This option will put pages in single files (per memseg list), as opposed to creating a file per page. This is normally not needed, but can be useful for use cases like userspace vhost, where there is limited number of page file descriptors that can be passed to VirtIO.

If the application (or DPDK-internal code, such as device drivers) wishes to receive notifications about newly allocated memory, it is possible to register for memory event callbacks via `rte_mem_event_callback_register()` function. This will call a callback function any time DPDK's memory map has changed.

If the application (or DPDK-internal code, such as device drivers) wishes to be notified about memory allocations above specified threshold (and have a chance to deny them), allocation validator callbacks are also available via `rte_mem_alloc_validator_callback_register()` function.

A default validator callback is provided by EAL, which can be enabled with a `--socket-limit` command-line option, for a simple way to limit maximum amount of memory that can be used by DPDK application.

Warning: Memory subsystem uses DPDK IPC internally, so memory allocations/callbacks and IPC must not be mixed: it is not safe to allocate/free memory inside memory-related or IPC callbacks, and it is not safe to use IPC inside memory-related callbacks. See chapter *Multi-process Support* for more details about DPDK IPC.

- Legacy memory mode

This mode is enabled by specifying `--legacy-mem` command-line switch to the EAL. This switch will have no effect on FreeBSD as FreeBSD only supports legacy mode anyway.

This mode mimics historical behavior of EAL. That is, EAL will reserve all memory at startup, sort all memory into large IOVA-contiguous chunks, and will not allow acquiring or releasing hugepages from the system at runtime.

If neither `-m` nor `--socket-mem` were specified, the entire available hugepage memory will be preallocated.

- Hugepage allocation matching

This behavior is enabled by specifying the `--match-allocations` command-line switch to the EAL. This switch is Linux-only and not supported with `--legacy-mem` nor `--no-huge`.

Some applications using memory event callbacks may require that hugepages be freed exactly as they were allocated. These applications may also require that any allocation from the malloc heap not span across allocations associated with two different memory event callbacks. Hugepage allocation matching can be used by these types of applications to satisfy both of these requirements. This can result in some increased memory usage which is very dependent on the memory allocation patterns of the application.

- 32-bit support

Additional restrictions are present when running in 32-bit mode. In dynamic memory mode, by default maximum of 2 gigabytes of VA space will be preallocated, and all of it will be on master lcore NUMA node unless `--socket-mem` flag is used.

In legacy mode, VA space will only be preallocated for segments that were requested (plus padding, to keep IOVA-contiguousness).

- Maximum amount of memory

All possible virtual memory space that can ever be used for hugepage mapping in a DPDK process is preallocated at startup, thereby placing an upper limit on how much memory a DPDK application can have. DPDK memory is stored in segment lists, each segment is strictly one physical page. It is possible to change the amount of virtual memory being preallocated at startup by editing the following config variables:

- `CONFIG_RTE_MAX_MEMSEG_LISTS` controls how many segment lists can DPDK have
- `CONFIG_RTE_MAX_MEM_MB_PER_LIST` controls how much megabytes of memory each segment list can address
- `CONFIG_RTE_MAX_MEMSEG_PER_LIST` controls how many segments each segment can have
- `CONFIG_RTE_MAX_MEMSEG_PER_TYPE` controls how many segments each memory type can have (where "type" is defined as "page size + NUMA node" combination)
- `CONFIG_RTE_MAX_MEM_MB_PER_TYPE` controls how much megabytes of memory each memory type can address
- `CONFIG_RTE_MAX_MEM_MB` places a global maximum on the amount of memory DPDK can reserve

Normally, these options do not need to be changed.

Note: Preallocated virtual memory is not to be confused with preallocated hugepage memory! All DPDK processes preallocate virtual memory at startup. Hugepages can later be mapped into that preallocated VA space (if dynamic memory mode is enabled), and can optionally be mapped into it at startup.

- Segment file descriptors

On Linux, in most cases, EAL will store segment file descriptors in EAL. This can become a problem when using smaller page sizes due to underlying limitations of `glibc` library. For example, Linux API calls such as `select()` may not work correctly because `glibc` does not support more than certain number of file descriptors.

There are two possible solutions for this problem. The recommended solution is to use `--single-file-segments` mode, as that mode will not use a file descriptor per each page, and it will keep compatibility with Virtio with vhost-user backend. This option is not available when using `--legacy-mem` mode.

Another option is to use bigger page sizes. Since fewer pages are required to cover the same memory area, fewer file descriptors will be stored internally by EAL.

Support for Externally Allocated Memory

It is possible to use externally allocated memory in DPDK. There are two ways in which using externally allocated memory can work: the malloc heap API's, and manual memory management.

- Using heap API's for externally allocated memory

Using a set of malloc heap API's is the recommended way to use externally allocated memory in DPDK. In this way, support for externally allocated memory is implemented through overloading the socket ID - externally allocated heaps will have socket ID's that would be considered invalid under normal circumstances. Requesting an allocation to take place from a specified externally allocated memory is a matter of supplying the correct socket ID to DPDK allocator, either directly (e.g. through a call to `rte_malloc`) or indirectly (through data structure-specific allocation API's such as `rte_ring_create`). Using these API's also ensures that mapping of externally allocated memory for DMA is also performed on any memory segment that is added to a DPDK malloc heap.

Since there is no way DPDK can verify whether memory is available or valid, this responsibility falls on the shoulders of the user. All multiprocess synchronization is also user's responsibility, as well as ensuring that all calls to add/attach/detach/remove memory are done in the correct order. It is not required to attach to a memory area in all processes - only attach to memory areas as needed.

The expected workflow is as follows:

- Get a pointer to memory area
- Create a named heap
- **Add memory area(s) to the heap**
  - If IOVA table is not specified, IOVA addresses will be assumed to be unavailable, and DMA mappings will not be performed
  - Other processes must attach to the memory area before they can use it
- Get socket ID used for the heap
- Use normal DPDK allocation procedures, using supplied socket ID
- **If memory area is no longer needed, it can be removed from the heap**
  - Other processes must detach from this memory area before it can be removed
- **If heap is no longer needed, remove it**
  - Socket ID will become invalid and will not be reused

For more information, please refer to `rte_malloc` API documentation, specifically the `rte_malloc_heap_*` family of function calls.

- Using externally allocated memory without DPDK API's

While using heap API's is the recommended method of using externally allocated memory in DPDK, there are certain use cases where the overhead of DPDK heap API is undesirable - for example, when manual memory management is performed on an externally allocated area. To support use cases where externally allocated memory will not be used as part of normal DPDK workflow, there is also another set of API's under the `rte_extmem_*` namespace.

These API's are (as their name implies) intended to allow registering or unregistering externally allocated memory to/from DPDK's internal page table, to allow API's like `rte_mem_virt2memseg` etc. to work with externally allocated memory. Memory added this way will not be available for any regular DPDK allocators; DPDK will leave this memory for the user application to manage.

The expected workflow is as follows:

- Get a pointer to memory area
- **Register memory within DPDK**
  - If IOVA table is not specified, IOVA addresses will be assumed to be unavailable
  - Other processes must attach to the memory area before they can use it
- Perform DMA mapping with `rte_dev_dma_map` if needed
- Use the memory area in your application
- **If memory area is no longer needed, it can be unregistered**
  - If the area was mapped for DMA, unmapping must be performed before unregistering memory
  - Other processes must detach from the memory area before it can be unregistered

Since these externally allocated memory areas will not be managed by DPDK, it is therefore up to the user application to decide how to use them and what to do with them once they're registered.

Per-lcore and Shared Variables

Note: lcore refers to a logical execution unit of the processor, sometimes called a hardware *thread*.

Shared variables are the default behavior. Per-lcore variables are implemented using *Thread Local Storage* (TLS) to provide per-thread local storage.

Logs

A logging API is provided by EAL. By default, in a Linux application, logs are sent to syslog and also to the console. However, the log function can be overridden by the user to use a different logging mechanism.

Trace and Debug Functions

There are some debug functions to dump the stack in glibc. The rte_panic() function can voluntarily provoke a SIG_ABORT, which can trigger the generation of a core file, readable by gdb.

CPU Feature Identification

The EAL can query the CPU at runtime (using the rte_cpu_get_features() function) to determine which CPU features are available.

User Space Interrupt Event

- User Space Interrupt and Alarm Handling in Host Thread

The EAL creates a host thread to poll the UIO device file descriptors to detect the interrupts. Callbacks can be registered or unregistered by the EAL functions for a specific interrupt event and are called in the host thread asynchronously. The EAL also allows timed callbacks to be used in the same way as for NIC interrupts.

Note: In DPDK PMD, the only interrupts handled by the dedicated host thread are those for link status change (link up and link down notification) and for sudden device removal.

- RX Interrupt Event

The receive and transmit routines provided by each PMD don't limit themselves to execute in polling thread mode. To ease the idle polling with tiny throughput, it's useful to pause the polling and wait until the wake-up event happens. The RX interrupt is the first choice to be such kind of wake-up event, but probably won't be the only one.

EAL provides the event APIs for this event-driven thread mode. Taking Linux as an example, the implementation relies on epoll. Each thread can monitor an epoll instance in which all the wake-up events' file descriptors are added. The event file descriptors are created and mapped to the interrupt vectors according to the UIO/VFIO spec. From FreeBSD's perspective, kqueue is the alternative way, but not implemented yet.

EAL initializes the mapping between event file descriptors and interrupt vectors, while each device initializes the mapping between interrupt vectors and queues. In this way, EAL actually is unaware of the interrupt cause on the specific vector. The eth_dev driver takes responsibility to program the latter mapping.

Note: Per queue RX interrupt event is only allowed in VFIO which supports multiple MSI-X vector. In UIO, the RX interrupt together with other interrupt causes shares the same vector. In this case, when RX interrupt and LSC(link status change) interrupt are both enabled(intr_conf.lsc == 1 && intr_conf.rxq == 1), only the former is capable.

The RX interrupt are controlled/enabled/disabled by ethdev APIs - 'rte_eth_dev_rx_intr_*'. They return failure if the PMD hasn't support them yet. The intr_conf.rxq flag is used to turn on the capability of RX interrupt per device.

- Device Removal Event

This event is triggered by a device being removed at a bus level. Its underlying resources may have been made unavailable (i.e. PCI mappings unmapped). The PMD must make sure that on such occurrence, the application can still safely use its callbacks.

This event can be subscribed to in the same way one would subscribe to a link status change event. The execution context is thus the same, i.e. it is the dedicated interrupt host thread.

Considering this, it is likely that an application would want to close a device having emitted a Device Removal Event. In such case, calling `rte_eth_dev_close()` can trigger it to unregister its own Device Removal Event callback. Care must be taken not to close the device from the interrupt handler context. It is necessary to reschedule such closing operation.

Blacklisting

The EAL PCI device blacklist functionality can be used to mark certain NIC ports as blacklisted, so they are ignored by the DPDK. The ports to be blacklisted are identified using the PCIe* description (Domain:Bus:Device.Function).

Misc Functions

Locks and atomic operations are per-architecture (i686 and x86_64).

IOVA Mode Detection

IOVA Mode is selected by considering what the current usable Devices on the system require and/or support.

On FreeBSD, RTE_IOVA_PA is always the default. On Linux, the IOVA mode is detected based on a 2-step heuristic detailed below.

For the first step, EAL asks each bus its requirement in terms of IOVA mode and decides on a preferred IOVA mode.

- if all buses report RTE_IOVA_PA, then the preferred IOVA mode is RTE_IOVA_PA,
- if all buses report RTE_IOVA_VA, then the preferred IOVA mode is RTE_IOVA_VA,
- if all buses report RTE_IOVA_DC, no bus expressed a preferrence, then the preferred mode is RTE_IOVA_DC,
- if the buses disagree (at least one wants RTE_IOVA_PA and at least one wants RTE_IOVA_VA), then the preferred IOVA mode is RTE_IOVA_DC (see below with the check on Physical Addresses availability),

If the buses have expressed no preference on which IOVA mode to pick, then a default is selected using the following logic:

- if physical addresses are not available, RTE_IOVA_VA mode is used
- if /sys/kernel/iommu_groups is not empty, RTE_IOVA_VA mode is used
- otherwise, RTE_IOVA_PA mode is used

In the case when the buses had disagreed on their preferred IOVA mode, part of the buses won't work because of this decision.

The second step checks if the preferred mode complies with the Physical Addresses availability since those are only available to root user in recent kernels. Namely, if the preferred mode is RTE_IOVA_PA but there is no access to Physical Addresses, then EAL init fails early, since later probing of the devices would fail anyway.

Note: The RTE_IOVA_VA mode is preferred as the default in most cases for the following reasons:

- All drivers are expected to work in RTE_IOVA_VA mode, irrespective of physical address availability.
- By default, the mempool, first asks for IOVA-contiguous memory using `RTE_MEMZONE_IOVA_CONTIG`. This is slow in RTE_IOVA_PA mode and it may affect the application boot time.
- It is easy to enable large amount of IOVA-contiguous memory use-cases with IOVA in VA mode.

It is expected that all PCI drivers work in both RTE_IOVA_PA and RTE_IOVA_VA modes.

If a PCI driver does not support RTE_IOVA_PA mode, the `RTE_PCI_DRV_NEED_IOVA_AS_VA` flag is used to dictate that this PCI driver can only work in RTE_IOVA_VA mode.

When the KNI kernel module is detected, RTE_IOVA_PA mode is preferred as a performance penalty is expected in RTE_IOVA_VA mode.

IOVA Mode Configuration

Auto detection of the IOVA mode, based on probing the bus and IOMMU configuration, may not report the desired addressing mode when virtual devices that are not directly attached to the bus are present. To facilitate forcing the IOVA mode to a specific value the EAL command line option `--iova-mode` can be used to select either physical addressing('pa') or virtual addressing('va').

5.3.2 Memory Segments and Memory Zones (memzone)

The mapping of physical memory is provided by this feature in the EAL. As physical memory can have gaps, the memory is described in a table of descriptors, and each descriptor (called rte_memseg) describes a physical page.

On top of this, the memzone allocator's role is to reserve contiguous portions of physical memory. These zones are identified by a unique name when the memory is reserved.

The rte_memzone descriptors are also located in the configuration structure. This structure is accessed using rte_eal_get_configuration(). The lookup (by name) of a memory zone returns a descriptor containing the physical address of the memory zone.

Memory zones can be reserved with specific start address alignment by supplying the align parameter (by default, they are aligned to cache line size). The alignment value should be a power of two and not less than the cache line size (64 bytes). Memory zones can also be reserved from either 2 MB or 1 GB hugepages, provided that both are available on the system.

Both memsegs and memzones are stored using `rte_fbarray` structures. Please refer to *DPDK API Reference* for more information.

5.3.3 Multiple pthread

DPDK usually pins one pthread per core to avoid the overhead of task switching. This allows for significant performance gains, but lacks flexibility and is not always efficient.

Power management helps to improve the CPU efficiency by limiting the CPU runtime frequency. However, alternately it is possible to utilize the idle cycles available to take advantage of the full capability of the CPU.

By taking advantage of cgroup, the CPU utilization quota can be simply assigned. This gives another way to improve the CPU efficiency, however, there is a prerequisite; DPDK must handle the context switching between multiple pthreads per core.

For further flexibility, it is useful to set pthread affinity not only to a CPU but to a CPU set.

EAL pthread and lcore Affinity

The term "lcore" refers to an EAL thread, which is really a Linux/FreeBSD pthread. "EAL pthreads" are created and managed by EAL and execute the tasks issued by *remote_launch*. In each EAL pthread, there is a TLS (Thread Local Storage) called *_lcore_id* for unique identification. As EAL pthreads usually bind 1:1 to the physical CPU, the *_lcore_id* is typically equal to the CPU ID.

When using multiple pthreads, however, the binding is no longer always 1:1 between an EAL pthread and a specified physical CPU. The EAL pthread may have affinity to a CPU set, and as such the *_lcore_id* will not be the same as the CPU ID. For this reason, there is an EAL long option '–lcores' defined to assign the CPU affinity of lcores. For a specified lcore ID or ID group, the option allows setting the CPU set for that EAL pthread.

The format pattern:

–lcores='<lcore_set>[@cpu_set][,<lcore_set>[@cpu_set],...]'

'lcore_set' and 'cpu_set' can be a single number, range or a group.

A number is a "digit([0-9]+)"; a range is "<number>-<number>"; a group is "(<number|range>[,<number|range>,...])".

If a '@cpu_set' value is not supplied, the value of 'cpu_set' will default to the value of 'lcore_set'.

```
For example, "--lcores='1,2@(5-7),(3-5)@(0,2),(0,6),7-8'" which means start 9 EAL␣
↪thread;
    lcore 0 runs on cpuset 0x41 (cpu 0,6);
    lcore 1 runs on cpuset 0x2 (cpu 1);
    lcore 2 runs on cpuset 0xe0 (cpu 5,6,7);
    lcore 3,4,5 runs on cpuset 0x5 (cpu 0,2);
    lcore 6 runs on cpuset 0x41 (cpu 0,6);
    lcore 7 runs on cpuset 0x80 (cpu 7);
    lcore 8 runs on cpuset 0x100 (cpu 8).
```

Using this option, for each given lcore ID, the associated CPUs can be assigned. It's also compatible with the pattern of corelist('-l') option.

non-EAL pthread support

It is possible to use the DPDK execution context with any user pthread (aka. Non-EAL pthreads). In a non-EAL pthread, the *_lcore_id* is always LCORE_ID_ANY which identifies that it is not an EAL thread with a valid, unique, *_lcore_id*. Some libraries will use an alternative unique ID (e.g. TID), some will not be impacted at all, and some will work but with limitations (e.g. timer and mempool libraries).

All these impacts are mentioned in *Known Issues* section.

Public Thread API

There are two public APIs `rte_thread_set_affinity()` and `rte_thread_get_affinity()` introduced for threads. When they're used in any pthread context, the Thread Local Storage(TLS) will be set/get.

Those TLS include *_cpuset* and *_socket_id*:

- *_cpuset* stores the CPUs bitmap to which the pthread is affinitized.
- *_socket_id* stores the NUMA node of the CPU set. If the CPUs in CPU set belong to different NUMA node, the *_socket_id* will be set to SOCKET_ID_ANY.

Control Thread API

It is possible to create Control Threads using the public API `rte_ctrl_thread_create()`. Those threads can be used for management/infrastructure tasks and are used internally by DPDK for multi process support and interrupt handling.

Those threads will be scheduled on CPUs part of the original process CPU affinity from which the dataplane and service lcores are excluded.

For example, on a 8 CPUs system, starting a dpdk application with -l 2,3 (dataplane cores), then depending on the affinity configuration which can be controlled with tools like taskset (Linux) or cpuset (FreeBSD),

- with no affinity configuration, the Control Threads will end up on 0-1,4-7 CPUs.
- with affinity restricted to 2-4, the Control Threads will end up on CPU 4.
- with affinity restricted to 2-3, the Control Threads will end up on CPU 2 (master lcore, which is the default when no CPU is available).

Known Issues

- rte_mempool

  The rte_mempool uses a per-lcore cache inside the mempool. For non-EAL pthreads, `rte_lcore_id()` will not return a valid number. So for now, when rte_mempool is used with non-EAL pthreads, the put/get operations will bypass the default mempool cache and there is a performance penalty because of this bypass. Only user-owned external caches can be used in a non-EAL context in conjunction with `rte_mempool_generic_put()` and `rte_mempool_generic_get()` that accept an explicit cache parameter.

- rte_ring

  rte_ring supports multi-producer enqueue and multi-consumer dequeue. However, it is non-preemptive, this has a knock on effect of making rte_mempool non-preemptable.

  Note: The "non-preemptive" constraint means:

  - a pthread doing multi-producers enqueues on a given ring must not be preempted by another pthread doing a multi-producer enqueue on the same ring.
  - a pthread doing multi-consumers dequeues on a given ring must not be preempted by another pthread doing a multi-consumer dequeue on the same ring.

  Bypassing this constraint may cause the 2nd pthread to spin until the 1st one is scheduled again. Moreover, if the 1st pthread is preempted by a context that has an higher priority, it may even cause a dead lock.

  This means, use cases involving preemptible pthreads should consider using rte_ring carefully.

  1. It CAN be used for preemptible single-producer and single-consumer use case.
  2. It CAN be used for non-preemptible multi-producer and preemptible single-consumer use case.
  3. It CAN be used for preemptible single-producer and non-preemptible multi-consumer use case.
  4. It MAY be used by preemptible multi-producer and/or preemptible multi-consumer pthreads whose scheduling policy are all SCHED_OTHER(cfs), SCHED_IDLE or SCHED_BATCH. User SHOULD be aware of the performance penalty before using it.
  5. It MUST not be used by multi-producer/consumer pthreads, whose scheduling policies are SCHED_FIFO or SCHED_RR.

  Alternatively, applications can use the lock-free stack mempool handler. When considering this handler, note that:

  - It is currently limited to the aarch64 and x86_64 platforms, because it uses an instruction (16-byte compare-and-swap) that is not yet available on other platforms.
  - It has worse average-case performance than the non-preemptive rte_ring, but software caching (e.g. the mempool cache) can mitigate this by reducing the number of stack accesses.

- rte_timer

  Running `rte_timer_manage()` on a non-EAL pthread is not allowed. However, resetting/stopping the timer from a non-EAL pthread is allowed.

- rte_log

  In non-EAL pthreads, there is no per thread loglevel and logtype, global loglevels are used.

- misc

  The debug statistics of rte_ring, rte_mempool and rte_timer are not supported in a non-EAL pthread.

cgroup control

The following is a simple example of cgroup control usage, there are two pthreads(t0 and t1) doing packet I/O on the same core ($CPU). We expect only 50% of CPU spend on packet IO.

```
mkdir /sys/fs/cgroup/cpu/pkt_io
mkdir /sys/fs/cgroup/cpuset/pkt_io

echo $cpu > /sys/fs/cgroup/cpuset/cpuset.cpus

echo $t0 > /sys/fs/cgroup/cpu/pkt_io/tasks
echo $t0 > /sys/fs/cgroup/cpuset/pkt_io/tasks

echo $t1 > /sys/fs/cgroup/cpu/pkt_io/tasks
echo $t1 > /sys/fs/cgroup/cpuset/pkt_io/tasks

cd /sys/fs/cgroup/cpu/pkt_io
echo 100000 > pkt_io/cpu.cfs_period_us
echo  50000 > pkt_io/cpu.cfs_quota_us
```

5.3.4 Malloc

The EAL provides a malloc API to allocate any-sized memory.

The objective of this API is to provide malloc-like functions to allow allocation from hugepage memory and to facilitate application porting. The *DPDK API Reference* manual describes the available functions.

Typically, these kinds of allocations should not be done in data plane processing because they are slower than pool-based allocation and make use of locks within the allocation and free paths. However, they can be used in configuration code.

Refer to the rte_malloc() function description in the *DPDK API Reference* manual for more information.

Cookies

When CONFIG_RTE_MALLOC_DEBUG is enabled, the allocated memory contains overwrite protection fields to help identify buffer overflows.

Alignment and NUMA Constraints

The rte_malloc() takes an align argument that can be used to request a memory area that is aligned on a multiple of this value (which must be a power of two).

On systems with NUMA support, a call to the rte_malloc() function will return memory that has been allocated on the NUMA socket of the core which made the call. A set of APIs is also provided, to allow memory to be explicitly allocated on a NUMA socket directly, or by allocated on the NUMA socket where another core is located, in the case where the memory is to be used by a logical core other than on the one doing the memory allocation.

Use Cases

This API is meant to be used by an application that requires malloc-like functions at initialization time.

For allocating/freeing data at runtime, in the fast-path of an application, the memory pool library should be used instead.

Internal Implementation

Data Structures

There are two data structure types used internally in the malloc library:

- struct malloc_heap - used to track free space on a per-socket basis
- struct malloc_elem - the basic element of allocation and free-space tracking inside the library.

Structure: malloc_heap

The malloc_heap structure is used to manage free space on a per-socket basis. Internally, there is one heap structure per NUMA node, which allows us to allocate memory to a thread based on the NUMA node on which this thread runs. While this does not guarantee that the memory will be used on that NUMA node, it is no worse than a scheme where the memory is always allocated on a fixed or random node.

The key fields of the heap structure and their function are described below (see also diagram above):

- lock - the lock field is needed to synchronize access to the heap. Given that the free space in the heap is tracked using a linked list, we need a lock to prevent two threads manipulating the list at the same time.
- free_head - this points to the first element in the list of free nodes for this malloc heap.
- first - this points to the first element in the heap.
- last - this points to the last element in the heap.

Fig. 5.3: Example of a malloc heap and malloc elements within the malloc library

Structure: malloc_elem

The malloc_elem structure is used as a generic header structure for various blocks of memory. It is used in two different ways - all shown in the diagram above:

1. As a header on a block of free or allocated memory - normal case
2. As a padding header inside a block of memory

The most important fields in the structure and how they are used are described below.

Malloc heap is a doubly-linked list, where each element keeps track of its previous and next elements. Due to the fact that hugepage memory can come and go, neighboring malloc elements may not necessarily be adjacent in memory. Also, since a malloc element may span multiple pages, its contents

may not necessarily be IOVA-contiguous either - each malloc element is only guaranteed to be virtually contiguous.

Note: If the usage of a particular field in one of the above three usages is not described, the field can be assumed to have an undefined value in that situation, for example, for padding headers only the "state" and "pad" fields have valid values.

- heap - this pointer is a reference back to the heap structure from which this block was allocated. It is used for normal memory blocks when they are being freed, to add the newly-freed block to the heap's free-list.
- prev - this pointer points to previous header element/block in memory. When freeing a block, this pointer is used to reference the previous block to check if that block is also free. If so, and the two blocks are immediately adjacent to each other, then the two free blocks are merged to form a single larger block.
- next - this pointer points to next header element/block in memory. When freeing a block, this pointer is used to reference the next block to check if that block is also free. If so, and the two blocks are immediately adjacent to each other, then the two free blocks are merged to form a single larger block.
- free_list - this is a structure pointing to previous and next elements in this heap's free list. It is only used in normal memory blocks; on `malloc()` to find a suitable free block to allocate and on `free()` to add the newly freed element to the free-list.
- state - This field can have one of three values: `FREE`, `BUSY` or `PAD`. The former two are to indicate the allocation state of a normal memory block and the latter is to indicate that the element structure is a dummy structure at the end of the start-of-block padding, i.e. where the start of the data within a block is not at the start of the block itself, due to alignment constraints. In that case, the pad header is used to locate the actual malloc element header for the block.
- pad - this holds the length of the padding present at the start of the block. In the case of a normal block header, it is added to the address of the end of the header to give the address of the start of the data area, i.e. the value passed back to the application on a malloc. Within a dummy header inside the padding, this same value is stored, and is subtracted from the address of the dummy header to yield the address of the actual block header.
- size - the size of the data block, including the header itself.

Memory Allocation

On EAL initialization, all preallocated memory segments are setup as part of the malloc heap. This setup involves placing an *element header* with `FREE` at the start of each virtually contiguous segment of memory. The `FREE` element is then added to the `free_list` for the malloc heap.

This setup also happens whenever memory is allocated at runtime (if supported), in which case newly allocated pages are also added to the heap, merging with any adjacent free segments if there are any.

When an application makes a call to a malloc-like function, the malloc function will first index the `lcore_config` structure for the calling thread, and determine the NUMA node of that thread. The NUMA node is used to index the array of `malloc_heap` structures which is passed as a parameter to the `heap_alloc()` function, along with the requested size, type, alignment and boundary parameters.

The `heap_alloc()` function will scan the free_list of the heap, and attempt to find a free block suitable for storing data of the requested size, with the requested alignment and boundary constraints.

When a suitable free element has been identified, the pointer to be returned to the user is calculated. The cache-line of memory immediately preceding this pointer is filled with a struct malloc_elem header. Because of alignment and boundary constraints, there could be free space at the start and/or end of the element, resulting in the following behavior:

1. Check for trailing space. If the trailing space is big enough, i.e. > 128 bytes, then the free element is split. If it is not, then we just ignore it (wasted space).
2. Check for space at the start of the element. If the space at the start is small, i.e. <=128 bytes, then a pad header is used, and the remaining space is wasted. If, however, the remaining space is greater, then the free element is split.

The advantage of allocating the memory from the end of the existing element is that no adjustment of the free list needs to take place - the existing element on the free list just has its size value adjusted, and the next/previous elements have their "prev"/"next" pointers redirected to the newly created element.

In case when there is not enough memory in the heap to satisfy allocation request, EAL will attempt to allocate more memory from the system (if supported) and, following successful allocation, will retry reserving the memory again. In a multiprocessing scenario, all primary and secondary processes will synchronize their memory maps to ensure that any valid pointer to DPDK memory is guaranteed to be valid at all times in all currently running processes.

Failure to synchronize memory maps in one of the processes will cause allocation to fail, even though some of the processes may have allocated the memory successfully. The memory is not added to the malloc heap unless primary process has ensured that all other processes have mapped this memory successfully.

Any successful allocation event will trigger a callback, for which user applications and other DPDK subsystems can register. Additionally, validation callbacks will be triggered before allocation if the newly allocated memory will exceed threshold set by the user, giving a chance to allow or deny allocation.

Note: Any allocation of new pages has to go through primary process. If the primary process is not active, no memory will be allocated even if it was theoretically possible to do so. This is because primary's process map acts as an authority on what should or should not be mapped, while each secondary process has its own, local memory map. Secondary processes do not update the shared memory map, they only copy its contents to their local memory map.

Freeing Memory

To free an area of memory, the pointer to the start of the data area is passed to the free function. The size of the `malloc_elem` structure is subtracted from this pointer to get the element header for the block. If this header is of type `PAD` then the pad length is further subtracted from the pointer to get the proper element header for the entire block.

From this element header, we get pointers to the heap from which the block was allocated and to where it must be freed, as well as the pointer to the previous and next elements. These next and previous elements are then checked to see if they are also `FREE` and are immediately adjacent to the current one, and if so, they are merged with the current element. This means that we can never have two `FREE` memory blocks adjacent to one another, as they are always merged into a single block.

If deallocating pages at runtime is supported, and the free element encloses one or more pages, those pages can be deallocated and be removed from the heap. If DPDK was started with command-line parameters for preallocating memory (`-m` or `--socket-mem`), then those pages that were allocated at startup will not be deallocated.

Any successful deallocation event will trigger a callback, for which user applications and other DPDK subsystems can register.

5.4 Service Cores

DPDK has a concept known as service cores, which enables a dynamic way of performing work on DPDK lcores. Service core support is built into the EAL, and an API is provided to optionally allow applications to control how the service cores are used at runtime.

The service cores concept is built up out of services (components of DPDK that require CPU cycles to operate) and service cores (DPDK lcores, tasked with running services). The power of the service core concept is that the mapping between service cores and services can be configured to abstract away the difference between platforms and environments.

For example, the Eventdev has hardware and software PMDs. Of these the software PMD requires an lcore to perform the scheduling operations, while the hardware PMD does not. With service cores, the application would not directly notice that the scheduling is done in software.

For detailed information about the service core API, please refer to the docs.

5.4.1 Service Core Initialization

There are two methods to having service cores in a DPDK application, either by using the service coremask, or by dynamically adding cores using the API. The simpler of the two is to pass the *-s* coremask argument to EAL, which will take any cores available in the main DPDK coremask, and if the bits are also set in the service coremask the cores become service-cores instead of DPDK application lcores.

5.4.2 Enabling Services on Cores

Each registered service can be individually mapped to a service core, or set of service cores. Enabling a service on a particular core means that the lcore in question will run the service. Disabling that core on the service stops the lcore in question from running the service.

Using this method, it is possible to assign specific workloads to each service core, and map N workloads to M number of service cores. Each service lcore loops over the services that are enabled for that core, and invokes the function to run the service.

5.4.3 Service Core Statistics

The service core library is capable of collecting runtime statistics like number of calls to a specific service, and number of cycles used by the service. The cycle count collection is dynamically configurable, allowing any application to profile the services running on the system at any time.

5.5 Trace Library

5.5.1 Overview

Tracing is a technique used to understand what goes on in a running software system. The software used for tracing is called a *tracer*, which is conceptually similar to a tape recorder. When recording, specific instrumentation points placed in the software source code generate events that are saved on a giant tape: a trace file. The trace file then later can be opened in *trace viewers* to visualize and analyze the trace events with timestamps and multi-core views. Such a mechanism will be useful for resolving a wide range of problems such as multi-core synchronization issues, latency measurements, finding out the post analysis information like CPU idle time, etc that would otherwise be extremely challenging to get.

Tracing is often compared to *logging*. However, tracers and loggers are two different tools, serving two different purposes. Tracers are designed to record much lower-level events that occur much more frequently than log messages, often in the range of thousands per second, with very little execution overhead. Logging is more appropriate for a very high-level analysis of less frequent events: user accesses, exceptional conditions (errors and warnings, for example), database transactions, instant messaging communications, and such. Simply put, logging is one of the many use cases that can be satisfied with tracing.

5.5.2 DPDK tracing library features

- A framework to add tracepoints in control and fast path APIs with minimum impact on performance. Typical trace overhead is ~20 cycles and instrumentation overhead is 1 cycle.
- Enable and disable the tracepoints at runtime.
- Save the trace buffer to the filesystem at any point in time.
- Support `overwrite` and `discard` trace mode operations.
- String-based tracepoint object lookup.
- Enable and disable a set of tracepoints based on regular expression and/or globbing.
- Generate trace in `Common Trace Format (CTF)`. `CTF` is an open-source trace format and is compatible with `LTTng`. For detailed information, refer to Common Trace Format.

5.5.3 How to add a tracepoint?

This section steps you through the details of adding a simple tracepoint.

Create the tracepoint header file

```
#include <rte_trace_point.h>

RTE_TRACE_POINT(
       app_trace_string,
       RTE_TRACE_POINT_ARGS(const char *str),
       rte_trace_point_emit_string(str);
)
```

The above macro creates `app_trace_string` tracepoint. The user can choose any name for the tracepoint. However, when adding a tracepoint in the DPDK library, the `rte_<library_name>_trace_[<domain>_]<name>` naming convention must be followed. The examples are `rte_eal_trace_generic_str`, `rte_mempool_trace_create`.

The `RTE_TRACE_POINT` macro expands from above definition as the following function template:

```
static __rte_always_inline void
app_trace_string(const char *str)
{
        /* Trace subsystem hooks */
        ...
        rte_trace_point_emit_string(str);
}
```

The consumer of this tracepoint can invoke `app_trace_string(const char *str)` to emit the trace event to the trace buffer.

Register the tracepoint

```
#include <rte_trace_point_register.h>

#include <my_tracepoint.h>

RTE_TRACE_POINT_DEFINE(app_trace_string);

RTE_INIT(app_trace_init)
{
      RTE_TRACE_POINT_REGISTER(app_trace_string, app.trace.string);
}
```

The above code snippet registers the `app_trace_string` tracepoint to trace library. Here, the `my_tracepoint.h` is the header file that the user created in the first step *Create the tracepoint header file*.

The second argument for the `RTE_TRACE_POINT_REGISTER` is the name for the tracepoint. This string will be used for tracepoint lookup or regular expression and/or glob based tracepoint operations. There is no requirement for the tracepoint function and its name to be similar. However, it is recommended to have a similar name for a better naming convention.

The user must register the tracepoint before the `rte_eal_init` invocation. The user can use the `RTE_INIT` construction scheme to achieve this.

Note: The `rte_trace_point_register.h` header must be included before any inclusion of the `rte_trace_point.h` header.

Note: The `RTE_TRACE_POINT_DEFINE` defines the placeholder for the `rte_trace_point_t` tracepoint object. The user must export a `__<trace_function_name>` symbol in the library `.map` file for this tracepoint to be used out of the library, in shared builds. For example, `__app_trace_string` will be the exported symbol in the above example.

5.5.4 Fast path tracepoint

In order to avoid performance impact in fast path code, the library introduced `RTE_TRACE_POINT_FP`. When adding the tracepoint in fast path code, the user must use `RTE_TRACE_POINT_FP` instead of `RTE_TRACE_POINT`.

`RTE_TRACE_POINT_FP` is compiled out by default and it can be enabled using `CONFIG_RTE_ENABLE_TRACE_FP` configuration parameter. The `enable_trace_fp` option shall be used for the same for meson build.

5.5.5 Event record mode

Event record mode is an attribute of trace buffers. Trace library exposes the following modes:

Overwrite
: When the trace buffer is full, new trace events overwrites the existing captured events in the trace buffer.

Discard
: When the trace buffer is full, new trace events will be discarded.

The mode can be configured either using EAL command line parameter `--trace-mode` on application boot up or use `rte_trace_mode_set()` API to configure at runtime.

5.5.6 Trace file location

On `rte_trace_save()` or `rte_eal_cleanup()` invocation, the library saves the trace buffers to the filesystem. By default, the trace files are stored in `$HOME/dpdk-traces/rte-yyyy-mm-dd-[AP]M-hh-mm-ss/`. It can be overridden by the `--trace-dir=<directory path>` EAL command line option.

For more information, refer to *EAL parameters* for trace EAL command line options.

5.5.7 View and analyze the recorded events

Once the trace directory is available, the user can view/inspect the recorded events.

There are many tools you can use to read DPDK traces:

1. `babeltrace` is a command-line utility that converts trace formats; it supports the format that DPDK trace library produces, CTF, as well as a basic text output that can be grep'ed. The babeltrace command is part of the Open Source Babeltrace project.

2. `Trace Compass` is a graphical user interface for viewing and analyzing any type of logs or traces, including DPDK traces.

Use the babeltrace command-line tool

The simplest way to list all the recorded events of a trace is to pass its path to babeltrace with no options:

```
babeltrace </path-to-trace-events/rte-yyyy-mm-dd-[AP]M-hh-mm-ss/>
```

`babeltrace` finds all traces recursively within the given path and prints all their events, merging them in chronological order.

You can pipe the output of the babeltrace into a tool like grep(1) for further filtering. Below example grep the events for `ethdev` only:

```
babeltrace /tmp/my-dpdk-trace | grep ethdev
```

You can pipe the output of babeltrace into a tool like wc(1) to count the recorded events. Below example count the number of `ethdev` events:

```
babeltrace /tmp/my-dpdk-trace | grep ethdev | wc --lines
```

Use the tracecompass GUI tool

`Tracecompass` is another tool to view/analyze the DPDK traces which gives a graphical view of events. Like `babeltrace`, tracecompass also provides an interface to search for a particular event. To use `tracecompass`, following are the minimum required steps:

- Install `tracecompass` to the localhost. Variants are available for Linux, Windows, and OS-X.
- Launch `tracecompass` which will open a graphical window with trace management interfaces.
- Open a trace using `File->Open Trace` option and select metadata file which is to be viewed/analyzed.

For more details, refer Trace Compass.

5.5.8 Quick start

This section steps you through the details of generating trace and viewing it.

- Start the dpdk-test:

```
echo "quit" | ./build/app/test/dpdk-test --no-huge --trace=.*
```

- View the traces with babeltrace viewer:

```
babeltrace $HOME/dpdk-traces/rte-yyyy-mm-dd-[AP]M-hh-mm-ss/
```

5.5.9 Implementation details

As DPDK trace library is designed to generate traces that uses `Common Trace Format (CTF)`. `CTF` specification consists of the following units to create a trace.

- `Stream` Sequence of packets.
- `Packet` Header and one or more events.
- `Event` Header and payload.

For detailed information, refer to Common Trace Format.

The implementation details broadly divided into the following areas:

Trace metadata creation

Based on the `CTF` specification, one of a CTF trace's streams is mandatory: the metadata stream. It contains exactly what you would expect: data about the trace itself. The metadata stream contains a textual description of the binary layouts of all the other streams.

This description is written using the Trace Stream Description Language (TSDL), a declarative language that exists only in the realm of CTF. The purpose of the metadata stream is to make CTF readers know how to parse a trace's binary streams of events without CTF specifying any fixed layout. The only stream layout known in advance is, in fact, the metadata stream's one.

The internal `trace_metadata_create()` function generates the metadata.

Trace memory

The trace memory will be allocated through an internal function `__rte_trace_mem_per_thread_alloc()`. The trace memory will be allocated per thread to enable lock less trace-emit function. The memory for the trace memory for DPDK lcores will be allocated on `rte_eal_init()` if the trace is enabled through a EAL option. For non DPDK threads, on the first trace emission, the memory will be allocated.

Trace memory layout

Table 5.1: Trace memory layout.

packet.header
packet.context
trace 0 header
trace 0 payload
trace 1 header
trace 1 payload
trace N header
trace N payload

packet.header

Table 5.2: Packet header layout.

uint32_t magic
rte_uuid_t uuid

packet.context

Table 5.3: Packet context layout.

uint32_t thread_id
char thread_name[32]

trace.header

Table 5.4: Trace header layout.

event_id [63:48]
timestamp [47:0]

The trace header is 64 bits, it consists of 48 bits of timestamp and 16 bits event ID.

The `packet.header` and `packet.context` will be written in the slow path at the time of trace memory creation. The `trace.header` and trace payload will be emitted when the tracepoint function is invoked.

5.6 RCU Library

Lockless data structures provide scalability and determinism. They enable use cases where locking may not be allowed (for example real-time applications).

In the following sections, the term "memory" refers to memory allocated by typical APIs like malloc() or anything that is representative of memory, for example an index of a free element array.

Since these data structures are lockless, the writers and readers are accessing the data structures concurrently. Hence, while removing an element from a data structure, the writers cannot return the memory to the allocator, without knowing that the readers are not referencing that element/memory anymore. Hence, it is required to separate the operation of removing an element into two steps:

1. Delete: in this step, the writer removes the reference to the element from the data structure but does not return the associated memory to the allocator. This will ensure that new readers will not get a reference to the removed element. Removing the reference is an atomic operation.
2. Free (Reclaim): in this step, the writer returns the memory to the memory allocator only after knowing that all the readers have stopped referencing the deleted element.

This library helps the writer determine when it is safe to free the memory by making use of thread Quiescent State (QS).

5.6.1 What is Quiescent State

Quiescent State can be defined as "any point in the thread execution where the thread does not hold a reference to shared memory". It is the responsibility of the application to determine its quiescent state.

Let us consider the following diagram:

Fig. 5.4: Phases in the Quiescent State model.

As shown in Fig. 5.4, reader thread 1 accesses data structures D1 and D2. When it is accessing D1, if the writer has to remove an element from D1, the writer cannot free the memory associated with that element immediately. The writer can return the memory to the allocator only after the reader stops referencing D1. In other words, reader thread RT1 has to enter a quiescent state.

Similarly, since reader thread 2 is also accessing D1, the writer has to wait till thread 2 enters quiescent state as well.

However, the writer does not need to wait for reader thread 3 to enter quiescent state. Reader thread 3 was not accessing D1 when the delete operation happened. So, reader thread 3 will not have a reference to the deleted entry.

It can be noted that, the critical sections for D2 is a quiescent state for D1. i.e. for a given data structure Dx, any point in the thread execution that does not reference Dx is a quiescent state.

Since memory is not freed immediately, there might be a need for provisioning of additional memory, depending on the application requirements.

5.6.2 Factors affecting the RCU mechanism

It is important to make sure that this library keeps the overhead of identifying the end of grace period and subsequent freeing of memory, to a minimum. The following paras explain how grace period and critical section affect this overhead.

The writer has to poll the readers to identify the end of grace period. Polling introduces memory accesses and wastes CPU cycles. The memory is not available for reuse during the grace period. Longer grace periods exasperate these conditions.

The length of the critical section and the number of reader threads is proportional to the duration of the grace period. Keeping the critical sections smaller will keep the grace period smaller. However, keeping the critical sections smaller requires additional CPU cycles (due to additional reporting) in the readers.

Hence, we need the characteristics of a small grace period and large critical section. This library addresses these characteristics by allowing the writer to do other work without having to block until the readers report their quiescent state.

5.6.3 RCU in DPDK

For DPDK applications, the beginning and end of a `while(1)` loop (where no references to shared data structures are kept) act as perfect quiescent states. This will combine all the shared data structure accesses into a single, large critical section which helps keep the overhead on the reader side to a minimum.

DPDK supports a pipeline model of packet processing and service cores. In these use cases, a given data structure may not be used by all the workers in the application. The writer has to wait only for the workers that use the data structure to report their quiescent state. To provide the required flexibility, this library has a concept of a QS variable. If required, the application can create one QS variable per data structure to help it track the end of grace period for each data structure. This helps keep the length of grace period to a minimum.

5.6.4 How to use this library

The application must allocate memory and initialize a QS variable.

Applications can call `rte_rcu_qsbr_get_memsize()` to calculate the size of memory to allocate. This API takes a maximum number of reader threads, using this variable, as a parameter.

Further, the application can initialize a QS variable using the API `rte_rcu_qsbr_init()`.

Each reader thread is assumed to have a unique thread ID. Currently, the management of the thread ID (for example allocation/free) is left to the application. The thread ID should be in the range of 0 to maximum number of threads provided while creating the QS variable. The application could also use `lcore_id` as the thread ID where applicable.

The `rte_rcu_qsbr_thread_register()` API will register a reader thread to report its quiescent state. This can be called from a reader thread. A control plane thread can also call this on behalf of a reader thread. The reader thread must call `rte_rcu_qsbr_thread_online()` API to start reporting its quiescent state.

Some of the use cases might require the reader threads to make blocking API calls (for example while using eventdev APIs). The writer thread should not wait for such reader threads to enter quiescent state. The reader thread must call `rte_rcu_qsbr_thread_offline()` API, before calling blocking APIs. It can call `rte_rcu_qsbr_thread_online()` API once the blocking API call returns.

The writer thread can trigger the reader threads to report their quiescent state by calling the API `rte_rcu_qsbr_start()`. It is possible for multiple writer threads to query the quiescent state status simultaneously. Hence, `rte_rcu_qsbr_start()` returns a token to each caller.

The writer thread must call `rte_rcu_qsbr_check()` API with the token to get the current quiescent state status. Option to block till all the reader threads enter the quiescent state is provided. If this API indicates that all the reader threads have entered the quiescent state, the application can free the deleted entry.

The APIs `rte_rcu_qsbr_start()` and `rte_rcu_qsbr_check()` are lock free. Hence, they can be called concurrently from multiple writers even while running as worker threads.

The separation of triggering the reporting from querying the status provides the writer threads flexibility to do useful work instead of blocking for the reader threads to enter the quiescent state or go offline. This reduces the memory accesses due to continuous polling for the status. But, since the resource is freed at a later time, the token and the reference to the deleted resource need to be stored for later queries.

The `rte_rcu_qsbr_synchronize()` API combines the functionality of `rte_rcu_qsbr_start()` and blocking `rte_rcu_qsbr_check()` into a single API. This API triggers the reader threads to report their quiescent state and polls till all the readers enter the quiescent state or go offline. This API does not allow the writer to do useful work while waiting and introduces additional memory accesses due to continuous polling. However, the application does not have to store the token or the reference to the deleted resource. The resource can be freed immediately after `rte_rcu_qsbr_synchronize()` API returns.

The reader thread must call `rte_rcu_qsbr_thread_offline()` and `rte_rcu_qsbr_thread_unregister()` APIs to remove itself from reporting its quiescent state. The `rte_rcu_qsbr_check()` API will not wait for this reader thread to report the quiescent state status anymore.

The reader threads should call `rte_rcu_qsbr_quiescent()` API to indicate that they entered a quiescent state. This API checks if a writer has triggered a quiescent state query and update the state accordingly.

The `rte_rcu_qsbr_lock()` and `rte_rcu_qsbr_unlock()` are empty functions. However, when `CONFIG_RTE_LIBRTE_RCU_DEBUG` is enabled, these APIs aid in debugging issues. One can mark the access to shared data structures on the reader side using these APIs. The `rte_rcu_qsbr_quiescent()` will check if all the locks are unlocked.

5.6.5 Resource reclamation framework for DPDK

Lock-free algorithms place additional burden of resource reclamation on the application. When a writer deletes an entry from a data structure, the writer:

1. Has to start the grace period
2. Has to store a reference to the deleted resources in a FIFO
3. Should check if the readers have completed a grace period and free the resources.

There are several APIs provided to help with this process. The writer can create a FIFO to store the references to deleted resources using `rte_rcu_qsbr_dq_create()`. The resources can be enqueued to this FIFO using `rte_rcu_qsbr_dq_enqueue()`. If the FIFO is full, `rte_rcu_qsbr_dq_enqueue` will reclaim the resources before enqueuing. It will also reclaim resources on regular basis to keep the FIFO from growing too large. If the writer runs out of resources, the writer can call `rte_rcu_qsbr_dq_reclaim` API to reclaim resources. `rte_rcu_qsbr_dq_delete` is provided to reclaim any remaining resources and free the FIFO while shutting down.

However, if this resource reclamation process were to be integrated in lock-free data structure libraries, it hides this complexity from the application and makes it easier for the application to adopt lock-free algorithms. The following paragraphs discuss how the reclamation process can be integrated in DPDK libraries.

In any DPDK application, the resource reclamation process using QSBR can be split into 4 parts:

1. Initialization
2. Quiescent State Reporting
3. Reclaiming Resources
4. Shutdown

The design proposed here assigns different parts of this process to client libraries and applications. The term 'client library' refers to lock-free data structure libraries such at rte_hash, rte_lpm etc. in DPDK or similar libraries outside of DPDK. The term 'application' refers to the packet processing application that makes use of DPDK such as L3 Forwarding example application, OVS, VPP etc..

The application has to handle 'Initialization' and 'Quiescent State Reporting'. So,

- the application has to create the RCU variable and register the reader threads to report their quiescent state.
- the application has to register the same RCU variable with the client library.
- reader threads in the application have to report the quiescent state. This allows for the application to control the length of the critical section/how frequently the application wants to report the quiescent state.

The client library will handle 'Reclaiming Resources' part of the process. The client libraries will make use of the writer thread context to execute the memory reclamation algorithm. So,

- client library should provide an API to register a RCU variable that it will use. It should call `rte_rcu_qsbr_dq_create()` to create the FIFO to store the references to deleted entries.
- client library should use `rte_rcu_qsbr_dq_enqueue` to enqueue the deleted resources on the FIFO and start the grace period.
- if the library runs out of resources while adding entries, it should call `rte_rcu_qsbr_dq_reclaim` to reclaim the resources and try the resource allocation again.

The 'Shutdown' process needs to be shared between the application and the client library.

- the application should make sure that the reader threads are not using the shared data structure, unregister the reader threads from the QSBR variable before calling the client library's shutdown function.
- client library should call `rte_rcu_qsbr_dq_delete` to reclaim any remaining resources and free the FIFO.

Integrating the resource reclamation with client libraries removes the burden from the application and makes it easy to use lock-free algorithms.

This design has several advantages over currently known methods.

1. Application does not need a dedicated thread to reclaim resources. Memory reclamation happens as part of the writer thread with little impact on performance.
2. The client library has better control over the resources. For example: the client library can attempt to reclaim when it has run out of resources.

5.7 Ring Library

The ring allows the management of queues. Instead of having a linked list of infinite size, the rte_ring has the following properties:

- FIFO
- Maximum size is fixed, the objects are stored in a table
- Objects can be pointers or elements of multiple of 4 byte size
- Lockless implementation
- Multi-consumer or single-consumer dequeue
- Multi-producer or single-producer enqueue
- Bulk dequeue - Dequeues the specified count of objects if successful; otherwise fails
- Bulk enqueue - Enqueues the specified count of objects if successful; otherwise fails
- Burst dequeue - Dequeue the maximum available objects if the specified count cannot be fulfilled
- Burst enqueue - Enqueue the maximum available objects if the specified count cannot be fulfilled

The advantages of this data structure over a linked list queue are as follows:

- Faster; only requires a single 32 bit Compare-And-Swap instruction instead of several pointer size Compare-And-Swap instructions.
- Simpler than a full lockless queue.
- Adapted to bulk enqueue/dequeue operations. As objects are stored in a table, a dequeue of several objects will not produce as many cache misses as in a linked queue. Also, a bulk dequeue of many objects does not cost more than a dequeue of a simple object.

The disadvantages:

- Size is fixed
- Having many rings costs more in terms of memory than a linked list queue. An empty ring contains at least N objects.

A simplified representation of a Ring is shown in with consumer and producer head and tail pointers to objects stored in the data structure.

Fig. 5.5: Ring Structure

5.7.1 References for Ring Implementation in FreeBSD*

The following code was added in FreeBSD 8.0, and is used in some network device drivers (at least in Intel drivers):

- bufring.h in FreeBSD
- bufring.c in FreeBSD

5.7.2 Lockless Ring Buffer in Linux*

The following is a link describing the Linux Lockless Ring Buffer Design.

5.7.3 Additional Features

Name

A ring is identified by a unique name. It is not possible to create two rings with the same name (rte_ring_create() returns NULL if this is attempted).

5.7.4 Use Cases

Use cases for the Ring library include:

- Communication between applications in the DPDK
- Used by memory pool allocator

5.7.5 Anatomy of a Ring Buffer

This section explains how a ring buffer operates. The ring structure is composed of two head and tail couples; one is used by producers and one is used by the consumers. The figures of the following sections refer to them as prod_head, prod_tail, cons_head and cons_tail.

Each figure represents a simplified state of the ring, which is a circular buffer. The content of the function local variables is represented on the top of the figure, and the content of ring structure is represented on the bottom of the figure.

Single Producer Enqueue

This section explains what occurs when a producer adds an object to the ring. In this example, only the producer head and tail (prod_head and prod_tail) are modified, and there is only one producer.

The initial state is to have a prod_head and prod_tail pointing at the same location.

Enqueue First Step

First, *ring->prod_head* and ring->cons_tail are copied in local variables. The prod_next local variable points to the next element of the table, or several elements after in case of bulk enqueue.

If there is not enough room in the ring (this is detected by checking cons_tail), it returns an error.

Fig. 5.6: Enqueue first step

Enqueue Second Step

The second step is to modify *ring->prod_head* in ring structure to point to the same location as prod_next.

The added object is copied in the ring (obj4).

Fig. 5.7: Enqueue second step

Enqueue Last Step

Once the object is added in the ring, ring->prod_tail in the ring structure is modified to point to the same location as *ring->prod_head*. The enqueue operation is finished.

Fig. 5.8: Enqueue last step

Single Consumer Dequeue

This section explains what occurs when a consumer dequeues an object from the ring. In this example, only the consumer head and tail (cons_head and cons_tail) are modified and there is only one consumer.

The initial state is to have a cons_head and cons_tail pointing at the same location.

Dequeue First Step

First, ring->cons_head and ring->prod_tail are copied in local variables. The cons_next local variable points to the next element of the table, or several elements after in the case of bulk dequeue.

If there are not enough objects in the ring (this is detected by checking prod_tail), it returns an error.

Fig. 5.9: Dequeue last step

Dequeue Second Step

The second step is to modify ring->cons_head in the ring structure to point to the same location as cons_next.

The dequeued object (obj1) is copied in the pointer given by the user.

Fig. 5.10: Dequeue second step

Dequeue Last Step

Finally, ring->cons_tail in the ring structure is modified to point to the same location as ring->cons_head. The dequeue operation is finished.

Fig. 5.11: Dequeue last step

Multiple Producers Enqueue

This section explains what occurs when two producers concurrently add an object to the ring. In this example, only the producer head and tail (prod_head and prod_tail) are modified.

The initial state is to have a prod_head and prod_tail pointing at the same location.

Multiple Producers Enqueue First Step

On both cores, *ring->prod_head* and ring->cons_tail are copied in local variables. The prod_next local variable points to the next element of the table, or several elements after in the case of bulk enqueue.

If there is not enough room in the ring (this is detected by checking cons_tail), it returns an error.

Fig. 5.12: Multiple producer enqueue first step

Multiple Producers Enqueue Second Step

The second step is to modify ring->prod_head in the ring structure to point to the same location as prod_next. This operation is done using a Compare And Swap (CAS) instruction, which does the following operations atomically:

- If ring->prod_head is different to local variable prod_head, the CAS operation fails, and the code restarts at first step.
- Otherwise, ring->prod_head is set to local prod_next, the CAS operation is successful, and processing continues.

In the figure, the operation succeeded on core 1, and step one restarted on core 2.

Fig. 5.13: Multiple producer enqueue second step

Multiple Producers Enqueue Third Step

The CAS operation is retried on core 2 with success.

The core 1 updates one element of the ring(obj4), and the core 2 updates another one (obj5).

Fig. 5.14: Multiple producer enqueue third step

Multiple Producers Enqueue Fourth Step

Each core now wants to update ring->prod_tail. A core can only update it if ring->prod_tail is equal to the prod_head local variable. This is only true on core 1. The operation is finished on core 1.

Fig. 5.15: Multiple producer enqueue fourth step

Multiple Producers Enqueue Last Step

Once ring->prod_tail is updated by core 1, core 2 is allowed to update it too. The operation is also finished on core 2.

Modulo 32-bit Indexes

In the preceding figures, the prod_head, prod_tail, cons_head and cons_tail indexes are represented by arrows. In the actual implementation, these values are not between 0 and size(ring)-1 as would be assumed. The indexes are between 0 and 2^32 -1, and we mask their value when we access the object table (the ring itself). 32-bit modulo also implies that operations on indexes (such as, add/subtract) will automatically do 2^32 modulo if the result overflows the 32-bit number range.

The following are two examples that help to explain how indexes are used in a ring.

Note: To simplify the explanation, operations with modulo 16-bit are used instead of modulo 32-bit. In addition, the four indexes are defined as unsigned 16-bit integers, as opposed to unsigned 32-bit integers in the more realistic case.

This ring contains 11000 entries.

This ring contains 12536 entries.

Note: For ease of understanding, we use modulo 65536 operations in the above examples. In real execution cases, this is redundant for low efficiency, but is done automatically when the result overflows.

Fig. 5.16: Multiple producer enqueue last step

Fig. 5.17: Modulo 32-bit indexes - Example 1

The code always maintains a distance between producer and consumer between 0 and size(ring)-1. Thanks to this property, we can do subtractions between 2 index values in a modulo-32bit base: that's why the overflow of the indexes is not a problem.

At any time, entries and free_entries are between 0 and size(ring)-1, even if only the first term of subtraction has overflowed:

```
uint32_t entries = (prod_tail - cons_head);
uint32_t free_entries = (mask + cons_tail -prod_head);
```

5.7.6 Producer/consumer synchronization modes

rte_ring supports different synchronization modes for producers and consumers. These modes can be specified at ring creation/init time via `flags` parameter. That should help users to configure ring in the most suitable way for his specific usage scenarios. Currently supported modes:

MP/MC (default one)

Multi-producer (/multi-consumer) mode. This is a default enqueue (/dequeue) mode for the ring. In this mode multiple threads can enqueue (/dequeue) objects to (/from) the ring. For 'classic' DPDK deployments (with one thread per core) this is usually the most suitable and fastest synchronization mode. As a well known limitation - it can perform quite pure on some overcommitted scenarios.

SP/SC

Single-producer (/single-consumer) mode. In this mode only one thread at a time is allowed to enqueue (/dequeue) objects to (/from) the ring.

MP_RTS/MC_RTS

Multi-producer (/multi-consumer) with Relaxed Tail Sync (RTS) mode. The main difference from the original MP/MC algorithm is that tail value is increased not by every thread that finished enqueue/dequeue, but only by the last one. That allows threads to avoid spinning on ring tail value, leaving actual tail value change to the last thread at a given instance. That technique helps to avoid the Lock-Waiter-Preemption (LWP) problem on tail update and improves average enqueue/dequeue times on overcommitted systems. To achieve that RTS requires 2 64-bit CAS for each enqueue(/dequeue) operation: one for head update, second for tail update. In comparison the original MP/MC algorithm requires one 32-bit CAS for head update and waiting/spinning on tail value.

Fig. 5.18: Modulo 32-bit indexes - Example 2

MP_HTS/MC_HTS

Multi-producer (/multi-consumer) with Head/Tail Sync (HTS) mode. In that mode enqueue/dequeue operation is fully serialized: at any given moment only one enqueue/dequeue operation can proceed. This is achieved by allowing a thread to proceed with changing `head.value` only when `head.value == tail.value`. Both head and tail values are updated atomically (as one 64-bit value). To achieve that 64-bit CAS is used by head update routine. That technique also avoids the Lock-Waiter-Preemption (LWP) problem on tail update and helps to improve ring enqueue/dequeue behavior in overcommitted scenarios. Another advantage of fully serialized producer/consumer - it provides the ability to implement MT safe peek API for rte_ring.

5.7.7 Ring Peek API

For ring with serialized producer/consumer (HTS sync mode) it is possible to split public enqueue/dequeue API into two phases:

- enqueue/dequeue start
- enqueue/dequeue finish

That allows user to inspect objects in the ring without removing them from it (aka MT safe peek) and reserve space for the objects in the ring before actual enqueue. Note that this API is available only for two sync modes:

- Single Producer/Single Consumer (SP/SC)
- Multi-producer/Multi-consumer with Head/Tail Sync (HTS)

It is a user responsibility to create/init ring with appropriate sync modes selected. As an example of usage:

```
/* read 1 elem from the ring: */
uint32_t n = rte_ring_dequeue_bulk_start(ring, &obj, 1, NULL);
if (n != 0) {
    /* examine object */
    if (object_examine(obj) == KEEP)
        /* decided to keep it in the ring. */
        rte_ring_dequeue_finish(ring, 0);
    else
        /* decided to remove it from the ring. */
        rte_ring_dequeue_finish(ring, n);
}
```

Note that between `_start_` and `_finish_` none other thread can proceed with enqueue(/dequeue) operation till `_finish_` completes.

5.7.8 References

- bufring.h in FreeBSD (version 8)
- bufring.c in FreeBSD (version 8)
- Linux Lockless Ring Buffer Design

5.8 Stack Library

DPDK's stack library provides an API for configuration and use of a bounded stack of pointers.

The stack library provides the following basic operations:

- Create a uniquely named stack of a user-specified size and using a user-specified socket, with either standard (lock-based) or lock-free behavior.
- Push and pop a burst of one or more stack objects (pointers). These function are multi-threading safe.
- Free a previously created stack.
- Lookup a pointer to a stack by its name.
- Query a stack's current depth and number of free entries.

5.8.1 Implementation

The library supports two types of stacks: standard (lock-based) and lock-free. Both types use the same set of interfaces, but their implementations differ.

Lock-based Stack

The lock-based stack consists of a contiguous array of pointers, a current index, and a spinlock. Accesses to the stack are made multi-thread safe by the spinlock.

Lock-free Stack

The lock-free stack consists of a linked list of elements, each containing a data pointer and a next pointer, and an atomic stack depth counter. The lock-free property means that multiple threads can push and pop simultaneously, and one thread being preempted/delayed in a push or pop operation will not impede the forward progress of any other thread.

The lock-free push operation enqueues a linked list of pointers by pointing the list's tail to the current stack head, and using a CAS to swing the stack head pointer to the head of the list. The operation retries if it is unsuccessful (i.e. the list changed between reading the head and modifying it), else it adjusts the stack length and returns.

The lock-free pop operation first reserves one or more list elements by adjusting the stack length, to ensure the dequeue operation will succeed without blocking. It then dequeues pointers by walking the list – starting from the head – then swinging the head pointer (using a CAS as well). While walking the list, the data pointers are recorded in an object table.

The linked list elements themselves are maintained in a lock-free LIFO, and are allocated before stack pushes and freed after stack pops. Since the stack has a fixed maximum depth, these elements do not need to be dynamically created.

The lock-free behavior is selected by passing the *RTE_STACK_F_LF* flag to rte_stack_create().

Preventing the ABA Problem

To prevent the ABA problem, this algorithm stack uses a 128-bit compare-and-swap instruction to atomically update both the stack top pointer and a modification counter. The ABA problem can occur without a modification counter if, for example:

1. Thread A reads head pointer X and stores the pointed-to list element.
2. Other threads modify the list such that the head pointer is once again X, but its pointed-to data is different than what thread A read.
3. Thread A changes the head pointer with a compare-and-swap and succeeds.

In this case thread A would not detect that the list had changed, and would both pop stale data and incorrect change the head pointer. By adding a modification counter that is updated on every push and pop as part of the compare-and-swap, the algorithm can detect when the list changes even if the head pointer remains the same.

5.9 Mempool Library

A memory pool is an allocator of a fixed-sized object. In the DPDK, it is identified by name and uses a mempool handler to store free objects. The default mempool handler is ring based. It provides some other optional services such as a per-core object cache and an alignment helper to ensure that objects are padded to spread them equally on all DRAM or DDR3 channels.

This library is used by the *Mbuf Library*.

5.9.1 Cookies

In debug mode (CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG is enabled), cookies are added at the beginning and end of allocated blocks. The allocated objects then contain overwrite protection fields to help debugging buffer overflows.

5.9.2 Stats

In debug mode (CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG is enabled), statistics about get from/put in the pool are stored in the mempool structure. Statistics are per-lcore to avoid concurrent access to statistics counters.

5.9.3 Memory Alignment Constraints on x86 architecture

Depending on hardware memory configuration on X86 architecture, performance can be greatly improved by adding a specific padding between objects. The objective is to ensure that the beginning of each object starts on a different channel and rank in memory so that all channels are equally loaded.

This is particularly true for packet buffers when doing L3 forwarding or flow classification. Only the first 64 bytes are accessed, so performance can be increased by spreading the start addresses of objects among the different channels.

The number of ranks on any DIMM is the number of independent sets of DRAMs that can be accessed for the full data bit-width of the DIMM. The ranks cannot be accessed simultaneously since they share the same data path. The physical layout of the DRAM chips on the DIMM itself does not necessarily relate to the number of ranks.

When running an application, the EAL command line options provide the ability to add the number of memory channels and ranks.

Note: The command line must always have the number of memory channels specified for the processor.

Examples of alignment for different DIMM architectures are shown in Fig. 5.19 and Fig. 5.20.

Fig. 5.19: Two Channels and Quad-ranked DIMM Example

In this case, the assumption is that a packet is 16 blocks of 64 bytes, which is not true.

The Intel® 5520 chipset has three channels, so in most cases, no padding is required between objects (except for objects whose size are n x 3 x 64 bytes blocks).

Fig. 5.20: Three Channels and Two Dual-ranked DIMM Example

When creating a new pool, the user can specify to use this feature or not.

5.9.4 Local Cache

In terms of CPU usage, the cost of multiple cores accessing a memory pool's ring of free buffers may be high since each access requires a compare-and-set (CAS) operation. To avoid having too many access requests to the memory pool's ring, the memory pool allocator can maintain a per-core cache and do bulk requests to the memory pool's ring, via the cache with many fewer locks on the actual memory pool structure. In this way, each core has full access to its own cache (with locks) of free objects and only when the cache fills does the core need to shuffle some of the free objects back to the pools ring or obtain more objects when the cache is empty.

While this may mean a number of buffers may sit idle on some core's cache, the speed at which a core can access its own cache for a specific memory pool without locks provides performance gains.

The cache is composed of a small, per-core table of pointers and its length (used as a stack). This internal cache can be enabled or disabled at creation of the pool.

The maximum size of the cache is static and is defined at compilation time (CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE).

Fig. 5.21: A mempool in Memory with its Associated Ring

Fig. 5.21 shows a cache in operation.

Alternatively to the internal default per-lcore local cache, an application can create and manage external caches through the `rte_mempool_cache_create()`, `rte_mempool_cache_free()` and `rte_mempool_cache_flush()` calls. These user-owned caches can be explicitly passed to `rte_mempool_generic_put()` and `rte_mempool_generic_get()`. The `rte_mempool_default_cache()` call returns the default internal cache if any. In contrast to the default caches, user-owned caches can be used by non-EAL threads too.

5.9.5 Mempool Handlers

This allows external memory subsystems, such as external hardware memory management systems and software based memory allocators, to be used with DPDK.

There are two aspects to a mempool handler.

- Adding the code for your new mempool operations (ops). This is achieved by adding a new mempool ops code, and using the `MEMPOOL_REGISTER_OPS` macro.
- Using the new API to call `rte_mempool_create_empty()` and `rte_mempool_set_ops_byname()` to create a new mempool and specifying which ops to use.

Several different mempool handlers may be used in the same application. A new mempool can be created by using the `rte_mempool_create_empty()` function, then using `rte_mempool_set_ops_byname()` to point the mempool to the relevant mempool handler callback (ops) structure.

Legacy applications may continue to use the old `rte_mempool_create()` API call, which uses a ring based mempool handler by default. These applications will need to be modified to use a new mempool handler.

For applications that use `rte_pktmbuf_create()`, there is a config setting (`RTE_MBUF_DEFAULT_MEMPOOL_OPS`) that allows the application to make use of an alternative mempool handler.

Note: When running a DPDK application with shared libraries, mempool handler shared objects specified with the '-d' EAL command-line parameter are dynamically loaded. When running a multi-process application with shared libraries, the -d arguments for mempool handlers *must be specified in the same order for all processes* to ensure correct operation.

5.9.6 Use Cases

All allocations that require a high level of performance should use a pool-based memory allocator. Below are some examples:

- *Mbuf Library*
- *Environment Abstraction Layer* , for logging service
- Any application that needs to allocate fixed-sized objects in the data plane and that will be continuously utilized by the system.

5.10 Mbuf Library

The mbuf library provides the ability to allocate and free buffers (mbufs) that may be used by the DPDK application to store message buffers. The message buffers are stored in a mempool, using the *Mempool Library*.

A rte_mbuf struct generally carries network packet buffers, but it can actually be any data (control data, events, ...). The rte_mbuf header structure is kept as small as possible and currently uses just two cache lines, with the most frequently used fields being on the first of the two cache lines.

5.10.1 Design of Packet Buffers

For the storage of the packet data (including protocol headers), two approaches were considered:

1. Embed metadata within a single memory buffer the structure followed by a fixed size area for the packet data.
2. Use separate memory buffers for the metadata structure and for the packet data.

The advantage of the first method is that it only needs one operation to allocate/free the whole memory representation of a packet. On the other hand, the second method is more flexible and allows the complete separation of the allocation of metadata structures from the allocation of packet data buffers.

The first method was chosen for the DPDK. The metadata contains control information such as message type, length, offset to the start of the data and a pointer for additional mbuf structures allowing buffer chaining.

Message buffers that are used to carry network packets can handle buffer chaining where multiple buffers are required to hold the complete packet. This is the case for jumbo frames that are composed of many mbufs linked together through their next field.

For a newly allocated mbuf, the area at which the data begins in the message buffer is RTE_PKTMBUF_HEADROOM bytes after the beginning of the buffer, which is cache aligned. Message buffers may be used to carry control information, packets, events, and so on between different entities in the system. Message buffers may also use their buffer pointers to point to other message buffer data sections or other structures.

Fig. 5.22 and Fig. 5.23 show some of these scenarios.

Fig. 5.22: An mbuf with One Segment

Fig. 5.23: An mbuf with Three Segments

The Buffer Manager implements a fairly standard set of buffer access functions to manipulate network packets.

5.10.2 Buffers Stored in Memory Pools

The Buffer Manager uses the *Mempool Library* to allocate buffers. Therefore, it ensures that the packet header is interleaved optimally across the channels and ranks for L3 processing. An mbuf contains a field indicating the pool that it originated from. When calling rte_pktmbuf_free(m), the mbuf returns to its original pool.

5.10.3 Constructors

Packet mbuf constructors are provided by the API. The rte_pktmbuf_init() function initializes some fields in the mbuf structure that are not modified by the user once created (mbuf type, origin pool, buffer start address, and so on). This function is given as a callback function to the rte_mempool_create() function at pool creation time.

5.10.4 Allocating and Freeing mbufs

Allocating a new mbuf requires the user to specify the mempool from which the mbuf should be taken. For any newly-allocated mbuf, it contains one segment, with a length of 0. The offset to data is initialized to have some bytes of headroom in the buffer (RTE_PKTMBUF_HEADROOM).

Freeing a mbuf means returning it into its original mempool. The content of an mbuf is not modified when it is stored in a pool (as a free mbuf). Fields initialized by the constructor do not need to be re-initialized at mbuf allocation.

When freeing a packet mbuf that contains several segments, all of them are freed and returned to their original mempool.

5.10.5 Manipulating mbufs

This library provides some functions for manipulating the data in a packet mbuf. For instance:

- Get data length
- Get a pointer to the start of data
- Prepend data before data
- Append data after data
- Remove data at the beginning of the buffer (rte_pktmbuf_adj())
- Remove data at the end of the buffer (rte_pktmbuf_trim()) Refer to the *DPDK API Reference* for details.

5.10.6 Meta Information

Some information is retrieved by the network driver and stored in an mbuf to make processing easier. For instance, the VLAN, the RSS hash result (see *Poll Mode Driver*) and a flag indicating that the checksum was computed by hardware.

An mbuf also contains the input port (where it comes from), and the number of segment mbufs in the chain.

For chained buffers, only the first mbuf of the chain stores this meta information.

For instance, this is the case on RX side for the IEEE1588 packet timestamp mechanism, the VLAN tagging and the IP checksum computation.

On TX side, it is also possible for an application to delegate some processing to the hardware if it supports it. For instance, the PKT_TX_IP_CKSUM flag allows to offload the computation of the IPv4 checksum.

The following examples explain how to configure different TX offloads on a vxlan-encapsulated tcp packet: `out_eth/out_ip/out_udp/vxlan/in_eth/in_ip/in_tcp/payload`

- calculate checksum of out_ip:

```
mb->l2_len = len(out_eth)
mb->l3_len = len(out_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM
set out_ip checksum to 0 in the packet
```

  This is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM.

- calculate checksum of out_ip and out_udp:

```
mb->l2_len = len(out_eth)
mb->l3_len = len(out_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM | PKT_TX_UDP_CKSUM
set out_ip checksum to 0 in the packet
set out_udp checksum to pseudo header using rte_ipv4_phdr_cksum()
```

  This is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM and DEV_TX_OFFLOAD_UDP_CKSUM.

- calculate checksum of in_ip:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM
set in_ip checksum to 0 in the packet
```

  This is similar to case 1), but l2_len is different. It is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM. Note that it can only work if outer L4 checksum is 0.

- calculate checksum of in_ip and in_tcp:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM | PKT_TX_TCP_CKSUM
set in_ip checksum to 0 in the packet
set in_tcp checksum to pseudo header using rte_ipv4_phdr_cksum()
```

  This is similar to case 2), but l2_len is different. It is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM and DEV_TX_OFFLOAD_TCP_CKSUM. Note that it

can only work if outer L4 checksum is 0.

- segment inner TCP:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->l4_len = len(in_tcp)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CKSUM | PKT_TX_TCP_CKSUM |
  PKT_TX_TCP_SEG;
set in_ip checksum to 0 in the packet
set in_tcp checksum to pseudo header without including the IP
  payload length using rte_ipv4_phdr_cksum()
```

This is supported on hardware advertising DEV_TX_OFFLOAD_TCP_TSO. Note that it can only work if outer L4 checksum is 0.

- calculate checksum of out_ip, in_ip, in_tcp:

```
mb->outer_l2_len = len(out_eth)
mb->outer_l3_len = len(out_ip)
mb->l2_len = len(out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_OUTER_IPV4 | PKT_TX_OUTER_IP_CKSUM  | \
  PKT_TX_IP_CKSUM |  PKT_TX_TCP_CKSUM;
set out_ip checksum to 0 in the packet
set in_ip checksum to 0 in the packet
set in_tcp checksum to pseudo header using rte_ipv4_phdr_cksum()
```

This is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM, DEV_TX_OFFLOAD_UDP_CKSUM and DEV_TX_OFFLOAD_OUTER_IPV4_CKSUM.

The list of flags and their precise meaning is described in the mbuf API documentation (rte_mbuf.h). Also refer to the testpmd source code (specifically the csumonly.c file) for details.

5.10.7 Direct and Indirect Buffers

A direct buffer is a buffer that is completely separate and self-contained. An indirect buffer behaves like a direct buffer but for the fact that the buffer pointer and data offset in it refer to data in another direct buffer. This is useful in situations where packets need to be duplicated or fragmented, since indirect buffers provide the means to reuse the same packet data across multiple buffers.

A buffer becomes indirect when it is "attached" to a direct buffer using the rte_pktmbuf_attach() function. Each buffer has a reference counter field and whenever an indirect buffer is attached to the direct buffer, the reference counter on the direct buffer is incremented. Similarly, whenever the indirect buffer is detached, the reference counter on the direct buffer is decremented. If the resulting reference counter is equal to 0, the direct buffer is freed since it is no longer in use.

There are a few things to remember when dealing with indirect buffers. First of all, an indirect buffer is never attached to another indirect buffer. Attempting to attach buffer A to indirect buffer B that is attached to C, makes rte_pktmbuf_attach() automatically attach A to C, effectively cloning B. Secondly, for a buffer to become indirect, its reference counter must be equal to 1, that is, it must not be already referenced by another indirect buffer. Finally, it is not possible to reattach an indirect buffer to the direct buffer (unless it is detached first).

While the attach/detach operations can be invoked directly using the recommended rte_pktmbuf_attach() and rte_pktmbuf_detach() functions, it is suggested to use the higher-level rte_pktmbuf_clone() function,

which takes care of the correct initialization of an indirect buffer and can clone buffers with multiple segments.

Since indirect buffers are not supposed to actually hold any data, the memory pool for indirect buffers should be configured to indicate the reduced memory consumption. Examples of the initialization of a memory pool for indirect buffers (as well as use case examples for indirect buffers) can be found in several of the sample applications, for example, the IPv4 Multicast sample application.

5.10.8 Debug

In debug mode (CONFIG_RTE_MBUF_DEBUG is enabled), the functions of the mbuf library perform sanity checks before any operation (such as, buffer corruption, bad type, and so on).

5.10.9 Use Cases

All networking application should use mbufs to transport network packets.

5.11 Poll Mode Driver

The DPDK includes 1 Gigabit, 10 Gigabit and 40 Gigabit and para virtualized virtio Poll Mode Drivers.

A Poll Mode Driver (PMD) consists of APIs, provided through the BSD driver running in user space, to configure the devices and their respective queues. In addition, a PMD accesses the RX and TX descriptors directly without any interrupts (with the exception of Link Status Change interrupts) to quickly receive, process and deliver packets in the user's application. This section describes the requirements of the PMDs, their global design principles and proposes a high-level architecture and a generic external API for the Ethernet PMDs.

5.11.1 Requirements and Assumptions

The DPDK environment for packet processing applications allows for two models, run-to-completion and pipe-line:

- In the *run-to-completion* model, a specific port's RX descriptor ring is polled for packets through an API. Packets are then processed on the same core and placed on a port's TX descriptor ring through an API for transmission.
- In the *pipe-line* model, one core polls one or more port's RX descriptor ring through an API. Packets are received and passed to another core via a ring. The other core continues to process the packet which then may be placed on a port's TX descriptor ring through an API for transmission.

In a synchronous run-to-completion model, each logical core assigned to the DPDK executes a packet processing loop that includes the following steps:

- Retrieve input packets through the PMD receive API
- Process each received packet one at a time, up to its forwarding
- Send pending output packets through the PMD transmit API

Conversely, in an asynchronous pipe-line model, some logical cores may be dedicated to the retrieval of received packets and other logical cores to the processing of previously received packets. Received

packets are exchanged between logical cores through rings. The loop for packet retrieval includes the following steps:

- Retrieve input packets through the PMD receive API
- Provide received packets to processing lcores through packet queues

The loop for packet processing includes the following steps:

- Retrieve the received packet from the packet queue
- Process the received packet, up to its retransmission if forwarded

To avoid any unnecessary interrupt processing overhead, the execution environment must not use any asynchronous notification mechanisms. Whenever needed and appropriate, asynchronous communication should be introduced as much as possible through the use of rings.

Avoiding lock contention is a key issue in a multi-core environment. To address this issue, PMDs are designed to work with per-core private resources as much as possible. For example, a PMD maintains a separate transmit queue per-core, per-port, if the PMD is not `DEV_TX_OFFLOAD_MT_LOCKFREE` capable. In the same way, every receive queue of a port is assigned to and polled by a single logical core (lcore).

To comply with Non-Uniform Memory Access (NUMA), memory management is designed to assign to each logical core a private buffer pool in local memory to minimize remote memory access. The configuration of packet buffer pools should take into account the underlying physical memory architecture in terms of DIMMS, channels and ranks. The application must ensure that appropriate parameters are given at memory pool creation time. See *Mempool Library*.

5.11.2 Design Principles

The API and architecture of the Ethernet* PMDs are designed with the following guidelines in mind.

PMDs must help global policy-oriented decisions to be enforced at the upper application level. Conversely, NIC PMD functions should not impede the benefits expected by upper-level global policies, or worse prevent such policies from being applied.

For instance, both the receive and transmit functions of a PMD have a maximum number of packets/descriptors to poll. This allows a run-to-completion processing stack to statically fix or to dynamically adapt its overall behavior through different global loop policies, such as:

- Receive, process immediately and transmit packets one at a time in a piecemeal fashion.
- Receive as many packets as possible, then process all received packets, transmitting them immediately.
- Receive a given maximum number of packets, process the received packets, accumulate them and finally send all accumulated packets to transmit.

To achieve optimal performance, overall software design choices and pure software optimization techniques must be considered and balanced against available low-level hardware-based optimization features (CPU cache properties, bus speed, NIC PCI bandwidth, and so on). The case of packet transmission is an example of this software/hardware tradeoff issue when optimizing burst-oriented network packet processing engines. In the initial case, the PMD could export only an rte_eth_tx_one function to transmit one packet at a time on a given queue. On top of that, one can easily build an rte_eth_tx_burst function that loops invoking the rte_eth_tx_one function to transmit several packets at a time. However, an rte_eth_tx_burst function is effectively implemented by the PMD to minimize the driver-level transmit cost per packet through the following optimizations:

- Share among multiple packets the un-amortized cost of invoking the rte_eth_tx_one function.
- Enable the rte_eth_tx_burst function to take advantage of burst-oriented hardware features (prefetch data in cache, use of NIC head/tail registers) to minimize the number of CPU cycles per packet, for example by avoiding unnecessary read memory accesses to ring transmit descriptors, or by systematically using arrays of pointers that exactly fit cache line boundaries and sizes.
- Apply burst-oriented software optimization techniques to remove operations that would otherwise be unavoidable, such as ring index wrap back management.

Burst-oriented functions are also introduced via the API for services that are intensively used by the PMD. This applies in particular to buffer allocators used to populate NIC rings, which provide functions to allocate/free several buffers at a time. For example, an mbuf_multiple_alloc function returning an array of pointers to rte_mbuf buffers which speeds up the receive poll function of the PMD when replenishing multiple descriptors of the receive ring.

5.11.3 Logical Cores, Memory and NIC Queues Relationships

The DPDK supports NUMA allowing for better performance when a processor's logical cores and interfaces utilize its local memory. Therefore, mbuf allocation associated with local PCIe* interfaces should be allocated from memory pools created in the local memory. The buffers should, if possible, remain on the local processor to obtain the best performance results and RX and TX buffer descriptors should be populated with mbufs allocated from a mempool allocated from local memory.

The run-to-completion model also performs better if packet or data manipulation is in local memory instead of a remote processors memory. This is also true for the pipe-line model provided all logical cores used are located on the same processor.

Multiple logical cores should never share receive or transmit queues for interfaces since this would require global locks and hinder performance.

If the PMD is `DEV_TX_OFFLOAD_MT_LOCKFREE` capable, multiple threads can invoke `rte_eth_tx_burst()` concurrently on the same tx queue without SW lock. This PMD feature found in some NICs and useful in the following use cases:

- Remove explicit spinlock in some applications where lcores are not mapped to Tx queues with 1:1 relation.
- In the eventdev use case, avoid dedicating a separate TX core for transmitting and thus enables more scaling as all workers can send the packets.

See *Hardware Offload* for `DEV_TX_OFFLOAD_MT_LOCKFREE` capability probing details.

5.11.4 Device Identification, Ownership and Configuration

Device Identification

Each NIC port is uniquely designated by its (bus/bridge, device, function) PCI identifiers assigned by the PCI probing/enumeration function executed at DPDK initialization. Based on their PCI identifier, NIC ports are assigned two other identifiers:

- A port index used to designate the NIC port in all functions exported by the PMD API.
- A port name used to designate the port in console messages, for administration or debugging purposes. For ease of use, the port name includes the port index.

Port Ownership

The Ethernet devices ports can be owned by a single DPDK entity (application, library, PMD, process, etc). The ownership mechanism is controlled by ethdev APIs and allows to set/remove/get a port owner by DPDK entities. Allowing this should prevent any multiple management of Ethernet port by different entities.

Note: It is the DPDK entity responsibility to set the port owner before using it and to manage the port usage synchronization between different threads or processes.

Device Configuration

The configuration of each NIC port includes the following operations:

- Allocate PCI resources
- Reset the hardware (issue a Global Reset) to a well-known default state
- Set up the PHY and the link
- Initialize statistics counters

The PMD API must also export functions to start/stop the all-multicast feature of a port and functions to set/unset the port in promiscuous mode.

Some hardware offload features must be individually configured at port initialization through specific configuration parameters. This is the case for the Receive Side Scaling (RSS) and Data Center Bridging (DCB) features for example.

On-the-Fly Configuration

All device features that can be started or stopped "on the fly" (that is, without stopping the device) do not require the PMD API to export dedicated functions for this purpose.

All that is required is the mapping address of the device PCI registers to implement the configuration of these features in specific functions outside of the drivers.

For this purpose, the PMD API exports a function that provides all the information associated with a device that can be used to set up a given device feature outside of the driver. This includes the PCI vendor identifier, the PCI device identifier, the mapping address of the PCI device registers, and the name of the driver.

The main advantage of this approach is that it gives complete freedom on the choice of the API used to configure, to start, and to stop such features.

As an example, refer to the configuration of the IEEE1588 feature for the Intel® 82576 Gigabit Ethernet Controller and the Intel® 82599 10 Gigabit Ethernet Controller controllers in the testpmd application.

Other features such as the L3/L4 5-Tuple packet filtering feature of a port can be configured in the same way. Ethernet* flow control (pause frame) can be configured on the individual port. Refer to the testpmd source code for details. Also, L4 (UDP/TCP/ SCTP) checksum offload by the NIC can be enabled for an individual packet as long as the packet mbuf is set up correctly. See *Hardware Offload* for details.

Configuration of Transmit Queues

Each transmit queue is independently configured with the following information:

- The number of descriptors of the transmit ring
- The socket identifier used to identify the appropriate DMA memory zone from which to allocate the transmit ring in NUMA architectures
- The values of the Prefetch, Host and Write-Back threshold registers of the transmit queue
- The *minimum* transmit packets to free threshold (tx_free_thresh). When the number of descriptors used to transmit packets exceeds this threshold, the network adaptor should be checked to see if it has written back descriptors. A value of 0 can be passed during the TX queue configuration to indicate the default value should be used. The default value for tx_free_thresh is 32. This ensures that the PMD does not search for completed descriptors until at least 32 have been processed by the NIC for this queue.
- The *minimum* RS bit threshold. The minimum number of transmit descriptors to use before setting the Report Status (RS) bit in the transmit descriptor. Note that this parameter may only be valid for Intel 10 GbE network adapters. The RS bit is set on the last descriptor used to transmit a packet if the number of descriptors used since the last RS bit setting, up to the first descriptor used to transmit the packet, exceeds the transmit RS bit threshold (tx_rs_thresh). In short, this parameter controls which transmit descriptors are written back to host memory by the network adapter. A value of 0 can be passed during the TX queue configuration to indicate that the default value should be used. The default value for tx_rs_thresh is 32. This ensures that at least 32 descriptors are used before the network adapter writes back the most recently used descriptor. This saves upstream PCIe* bandwidth resulting from TX descriptor write-backs. It is important to note that the TX Write-back threshold (TX wthresh) should be set to 0 when tx_rs_thresh is greater than 1. Refer to the Intel® 82599 10 Gigabit Ethernet Controller Datasheet for more details.

The following constraints must be satisfied for tx_free_thresh and tx_rs_thresh:

- tx_rs_thresh must be greater than 0.
- tx_rs_thresh must be less than the size of the ring minus 2.
- tx_rs_thresh must be less than or equal to tx_free_thresh.
- tx_free_thresh must be greater than 0.
- tx_free_thresh must be less than the size of the ring minus 3.
- For optimal performance, TX wthresh should be set to 0 when tx_rs_thresh is greater than 1.

One descriptor in the TX ring is used as a sentinel to avoid a hardware race condition, hence the maximum threshold constraints.

Note: When configuring for DCB operation, at port initialization, both the number of transmit queues and the number of receive queues must be set to 128.

Free Tx mbuf on Demand

Many of the drivers do not release the mbuf back to the mempool, or local cache, immediately after the packet has been transmitted. Instead, they leave the mbuf in their Tx ring and either perform a bulk release when the `tx_rs_thresh` has been crossed or free the mbuf when a slot in the Tx ring is needed.

An application can request the driver to release used mbufs with the `rte_eth_tx_done_cleanup()` API. This API requests the driver to release mbufs that are no longer in use, independent of whether or not the `tx_rs_thresh` has been crossed. There are two scenarios when an application may want the mbuf released immediately:

- When a given packet needs to be sent to multiple destination interfaces (either for Layer 2 flooding or Layer 3 multi-cast). One option is to make a copy of the packet or a copy of the header portion that needs to be manipulated. A second option is to transmit the packet and then poll the `rte_eth_tx_done_cleanup()` API until the reference count on the packet is decremented. Then the same packet can be transmitted to the next destination interface. The application is still responsible for managing any packet manipulations needed between the different destination interfaces, but a packet copy can be avoided. This API is independent of whether the packet was transmitted or dropped, only that the mbuf is no longer in use by the interface.
- Some applications are designed to make multiple runs, like a packet generator. For performance reasons and consistency between runs, the application may want to reset back to an initial state between each run, where all mbufs are returned to the mempool. In this case, it can call the `rte_eth_tx_done_cleanup()` API for each destination interface it has been using to request it to release of all its used mbufs.

To determine if a driver supports this API, check for the *Free Tx mbuf on demand* feature in the *Network Interface Controller Drivers* document.

Hardware Offload

Depending on driver capabilities advertised by `rte_eth_dev_info_get()`, the PMD may support hardware offloading feature like checksumming, TCP segmentation, VLAN insertion or lockfree multi-threaded TX burst on the same TX queue.

The support of these offload features implies the addition of dedicated status bit(s) and value field(s) into the rte_mbuf data structure, along with their appropriate handling by the receive/transmit functions exported by each PMD. The list of flags and their precise meaning is described in the mbuf API documentation and in the in *Mbuf Library*, section "Meta Information".

Per-Port and Per-Queue Offloads

In the DPDK offload API, offloads are divided into per-port and per-queue offloads as follows:

- A per-queue offloading can be enabled on a queue and disabled on another queue at the same time.
- A pure per-port offload is the one supported by device but not per-queue type.
- A pure per-port offloading can't be enabled on a queue and disabled on another queue at the same time.
- A pure per-port offloading must be enabled or disabled on all queues at the same time.
- Any offloading is per-queue or pure per-port type, but can't be both types at same devices.

- Port capabilities = per-queue capabilities + pure per-port capabilities.
- Any supported offloading can be enabled on all queues.

The different offloads capabilities can be queried using `rte_eth_dev_info_get()`. The `dev_info->[rt]x_queue_offload_capa` returned from `rte_eth_dev_info_get()` includes all per-queue offloading capabilities. The `dev_info->[rt]x_offload_capa` returned from `rte_eth_dev_info_get()` includes all pure per-port and per-queue offloading capabilities. Supported offloads can be either per-port or per-queue.

Offloads are enabled using the existing `DEV_TX_OFFLOAD_*` or `DEV_RX_OFFLOAD_*` flags. Any requested offloading by an application must be within the device capabilities. Any offloading is disabled by default if it is not set in the parameter `dev_conf->[rt]xmode.offloads` to `rte_eth_dev_configure()` and `[rt]x_conf->offloads` to `rte_eth_[rt]x_queue_setup()`.

If any offloading is enabled in `rte_eth_dev_configure()` by an application, it is enabled on all queues no matter whether it is per-queue or per-port type and no matter whether it is set or cleared in `[rt]x_conf->offloads` to `rte_eth_[rt]x_queue_setup()`.

If a per-queue offloading hasn't been enabled in `rte_eth_dev_configure()`, it can be enabled or disabled in `rte_eth_[rt]x_queue_setup()` for individual queue. A newly added offloads in `[rt]x_conf->offloads` to `rte_eth_[rt]x_queue_setup()` input by application is the one which hasn't been enabled in `rte_eth_dev_configure()` and is requested to be enabled in `rte_eth_[rt]x_queue_setup()`. It must be per-queue type, otherwise trigger an error log.

5.11.5 Poll Mode Driver API

Generalities

By default, all functions exported by a PMD are lock-free functions that are assumed not to be invoked in parallel on different logical cores to work on the same target object. For instance, a PMD receive function cannot be invoked in parallel on two logical cores to poll the same RX queue of the same port. Of course, this function can be invoked in parallel by different logical cores on different RX queues. It is the responsibility of the upper-level application to enforce this rule.

If needed, parallel accesses by multiple logical cores to shared queues can be explicitly protected by dedicated inline lock-aware functions built on top of their corresponding lock-free functions of the PMD API.

Generic Packet Representation

A packet is represented by an rte_mbuf structure, which is a generic metadata structure containing all necessary housekeeping information. This includes fields and status bits corresponding to offload hardware features, such as checksum computation of IP headers or VLAN tags.

The rte_mbuf data structure includes specific fields to represent, in a generic way, the offload features provided by network controllers. For an input packet, most fields of the rte_mbuf structure are filled in by the PMD receive function with the information contained in the receive descriptor. Conversely, for output packets, most fields of rte_mbuf structures are used by the PMD transmit function to initialize transmit descriptors.

The mbuf structure is fully described in the *Mbuf Library* chapter.

Ethernet Device API

The Ethernet device API exported by the Ethernet PMDs is described in the *DPDK API Reference*.

Ethernet Device Standard Device Arguments

Standard Ethernet device arguments allow for a set of commonly used arguments/ parameters which are applicable to all Ethernet devices to be available to for specification of specific device and for passing common configuration parameters to those ports.

- `representor` for a device which supports the creation of representor ports this argument allows user to specify which switch ports to enable port representors for.:

```
-w DBDF,representor=0
-w DBDF,representor=[0,4,6,9]
-w DBDF,representor=[0-31]
```

Note: PMDs are not required to support the standard device arguments and users should consult the relevant PMD documentation to see support devargs.

Extended Statistics API

The extended statistics API allows a PMD to expose all statistics that are available to it, including statistics that are unique to the device. Each statistic has three properties `name`, `id` and `value`:

- `name`: A human readable string formatted by the scheme detailed below.
- `id`: An integer that represents only that statistic.
- `value`: A unsigned 64-bit integer that is the value of the statistic.

Note that extended statistic identifiers are driver-specific, and hence might not be the same for different ports. The API consists of various `rte_eth_xstats_*()` functions, and allows an application to be flexible in how it retrieves statistics.

Scheme for Human Readable Names

A naming scheme exists for the strings exposed to clients of the API. This is to allow scraping of the API for statistics of interest. The naming scheme uses strings split by a single underscore `_`. The scheme is as follows:

- direction
- detail 1
- detail 2
- detail n
- unit

Examples of common statistics xstats strings, formatted to comply to the scheme proposed above:

- `rx_bytes`
- `rx_crc_errors`

- tx_multicast_packets

The scheme, although quite simple, allows flexibility in presenting and reading information from the statistic strings. The following example illustrates the naming scheme:rx_packets. In this example, the string is split into two components. The first component rx indicates that the statistic is associated with the receive side of the NIC. The second component packets indicates that the unit of measure is packets.

A more complicated example: tx_size_128_to_255_packets. In this example, tx indicates transmission, size is the first detail, 128 etc are more details, and packets indicates that this is a packet counter.

Some additions in the metadata scheme are as follows:

- If the first part does not match rx or tx, the statistic does not have an affinity with either receive of transmit.
- If the first letter of the second part is q and this q is followed by a number, this statistic is part of a specific queue.

An example where queue numbers are used is as follows: tx_q7_bytes which indicates this statistic applies to queue number 7, and represents the number of transmitted bytes on that queue.

API Design

The xstats API uses the name, id, and value to allow performant lookup of specific statistics. Performant lookup means two things;

- No string comparisons with the name of the statistic in fast-path
- Allow requesting of only the statistics of interest

The API ensures these requirements are met by mapping the name of the statistic to a unique id, which is used as a key for lookup in the fast-path. The API allows applications to request an array of id values, so that the PMD only performs the required calculations. Expected usage is that the application scans the name of each statistic, and caches the id if it has an interest in that statistic. On the fast-path, the integer can be used to retrieve the actual value of the statistic that the id represents.

API Functions

The API is built out of a small number of functions, which can be used to retrieve the number of statistics and the names, IDs and values of those statistics.

- rte_eth_xstats_get_names_by_id(): returns the names of the statistics. When given a NULL parameter the function returns the number of statistics that are available.
- rte_eth_xstats_get_id_by_name(): Searches for the statistic ID that matches xstat_name. If found, the id integer is set.
- rte_eth_xstats_get_by_id(): Fills in an array of uint64_t values with matching the provided ids array. If the ids array is NULL, it returns all statistics that are available.

Application Usage

Imagine an application that wants to view the dropped packet count. If no packets are dropped, the application does not read any other metrics for performance reasons. If packets are dropped, the application has a particular set of statistics that it requests. This "set" of statistics allows the app to decide what next steps to perform. The following code-snippets show how the xstats API can be used to achieve this goal.

First step is to get all statistics names and list them:

```
struct rte_eth_xstat_name *xstats_names;
uint64_t *values;
int len, i;

/* Get number of stats */
len = rte_eth_xstats_get_names_by_id(port_id, NULL, NULL, 0);
if (len < 0) {
    printf("Cannot get xstats count\n");
    goto err;
}

xstats_names = malloc(sizeof(struct rte_eth_xstat_name) * len);
if (xstats_names == NULL) {
    printf("Cannot allocate memory for xstat names\n");
    goto err;
}

/* Retrieve xstats names, passing NULL for IDs to return all statistics */
if (len != rte_eth_xstats_get_names_by_id(port_id, xstats_names, NULL, len)) {
    printf("Cannot get xstat names\n");
    goto err;
}

values = malloc(sizeof(values) * len);
if (values == NULL) {
    printf("Cannot allocate memory for xstats\n");
    goto err;
}

/* Getting xstats values */
if (len != rte_eth_xstats_get_by_id(port_id, NULL, values, len)) {
    printf("Cannot get xstat values\n");
    goto err;
}

/* Print all xstats names and values */
for (i = 0; i < len; i++) {
    printf("%s: %"PRIu64"\n", xstats_names[i].name, values[i]);
}
```

The application has access to the names of all of the statistics that the PMD exposes. The application can decide which statistics are of interest, cache the ids of those statistics by looking up the name as follows:

```
uint64_t id;
uint64_t value;
const char *xstat_name = "rx_errors";

if(!rte_eth_xstats_get_id_by_name(port_id, xstat_name, &id)) {
    rte_eth_xstats_get_by_id(port_id, &id, &value, 1);
    printf("%s: %"PRIu64"\n", xstat_name, value);
}
```

(continues on next page)

(continued from previous page)

```
else {
    printf("Cannot find xstats with a given name\n");
    goto err;
}
```

The API provides flexibility to the application so that it can look up multiple statistics using an array containing multiple `id` numbers. This reduces the function call overhead of retrieving statistics, and makes lookup of multiple statistics simpler for the application.

```
#define APP_NUM_STATS 4
/* application cached these ids previously; see above */
uint64_t ids_array[APP_NUM_STATS] = {3,4,7,21};
uint64_t value_array[APP_NUM_STATS];

/* Getting multiple xstats values from array of IDs */
rte_eth_xstats_get_by_id(port_id, ids_array, value_array, APP_NUM_STATS);

uint32_t i;
for(i = 0; i < APP_NUM_STATS; i++) {
    printf("%d: %"PRIu64"\n", ids_array[i], value_array[i]);
}
```

This array lookup API for xstats allows the application create multiple "groups" of statistics, and look up the values of those IDs using a single API call. As an end result, the application is able to achieve its goal of monitoring a single statistic ("rx_errors" in this case), and if that shows packets being dropped, it can easily retrieve a "set" of statistics using the IDs array parameter to `rte_eth_xstats_get_by_id` function.

NIC Reset API

```
int rte_eth_dev_reset(uint16_t port_id);
```

Sometimes a port has to be reset passively. For example when a PF is reset, all its VFs should also be reset by the application to make them consistent with the PF. A DPDK application also can call this function to trigger a port reset. Normally, a DPDK application would invokes this function when an RTE_ETH_EVENT_INTR_RESET event is detected.

It is the duty of the PMD to trigger RTE_ETH_EVENT_INTR_RESET events and the application should register a callback function to handle these events. When a PMD needs to trigger a reset, it can trigger an RTE_ETH_EVENT_INTR_RESET event. On receiving an RTE_ETH_EVENT_INTR_RESET event, applications can handle it as follows: Stop working queues, stop calling Rx and Tx functions, and then call rte_eth_dev_reset(). For thread safety all these operations should be called from the same thread.

For example when PF is reset, the PF sends a message to notify VFs of this event and also trigger an interrupt to VFs. Then in the interrupt service routine the VFs detects this notification message and calls _rte_eth_dev_callback_process(dev, RTE_ETH_EVENT_INTR_RESET, NULL). This means that a PF reset triggers an RTE_ETH_EVENT_INTR_RESET event within VFs. The function _rte_eth_dev_callback_process() will call the registered callback function. The callback function can trigger the application to handle all operations the VF reset requires including stopping Rx/Tx queues and calling rte_eth_dev_reset().

The rte_eth_dev_reset() itself is a generic function which only does some hardware reset operations through calling dev_unint() and dev_init(), and itself does not handle synchronization, which is handled by application.

The PMD itself should not call rte_eth_dev_reset(). The PMD can trigger the application to handle reset event. It is duty of application to handle all synchronization before it calls rte_eth_dev_reset().

5.12 Generic flow API (rte_flow)

5.12.1 Overview

This API provides a generic means to configure hardware to match specific ingress or egress traffic, alter its fate and query related counters according to any number of user-defined rules.

It is named *rte_flow* after the prefix used for all its symbols, and is defined in `rte_flow.h`.

- Matching can be performed on packet data (protocol headers, payload) and properties (e.g. associated physical port, virtual device function ID).
- Possible operations include dropping traffic, diverting it to specific queues, to virtual/physical device functions or ports, performing tunnel offloads, adding marks and so on.

It is slightly higher-level than the legacy filtering framework which it encompasses and supersedes (including all functions and filter types) in order to expose a single interface with an unambiguous behavior that is common to all poll-mode drivers (PMDs).

5.12.2 Flow rule

Description

A flow rule is the combination of attributes with a matching pattern and a list of actions. Flow rules form the basis of this API.

Flow rules can have several distinct actions (such as counting, encapsulating, decapsulating before redirecting packets to a particular queue, etc.), instead of relying on several rules to achieve this and having applications deal with hardware implementation details regarding their order.

Support for different priority levels on a rule basis is provided, for example in order to force a more specific rule to come before a more generic one for packets matched by both. However hardware support for more than a single priority level cannot be guaranteed. When supported, the number of available priority levels is usually low, which is why they can also be implemented in software by PMDs (e.g. missing priority levels may be emulated by reordering rules).

In order to remain as hardware-agnostic as possible, by default all rules are considered to have the same priority, which means that the order between overlapping rules (when a packet is matched by several filters) is undefined.

PMDs may refuse to create overlapping rules at a given priority level when they can be detected (e.g. if a pattern matches an existing filter).

Thus predictable results for a given priority level can only be achieved with non-overlapping rules, using perfect matching on all protocol layers.

Flow rules can also be grouped, the flow rule priority is specific to the group they belong to. All flow rules in a given group are thus processed within the context of that group. Groups are not linked by default, so the logical hierarchy of groups must be explicitly defined by flow rules themselves in each group using the JUMP action to define the next group to redirect too. Only flow rules defined in the default group 0

are guarantee to be matched against, this makes group 0 the origin of any group hierarchy defined by an application.

Support for multiple actions per rule may be implemented internally on top of non-default hardware priorities, as a result both features may not be simultaneously available to applications.

Considering that allowed pattern/actions combinations cannot be known in advance and would result in an impractically large number of capabilities to expose, a method is provided to validate a given rule from the current device configuration state.

This enables applications to check if the rule types they need is supported at initialization time, before starting their data path. This method can be used anytime, its only requirement being that the resources needed by a rule should exist (e.g. a target RX queue should be configured first).

Each defined rule is associated with an opaque handle managed by the PMD, applications are responsible for keeping it. These can be used for queries and rules management, such as retrieving counters or other data and destroying them.

To avoid resource leaks on the PMD side, handles must be explicitly destroyed by the application before releasing associated resources such as queues and ports.

The following sections cover:

- **Attributes** (represented by `struct rte_flow_attr`): properties of a flow rule such as its direction (ingress or egress) and priority.
- **Pattern item** (represented by `struct rte_flow_item`): part of a matching pattern that either matches specific packet data or traffic properties. It can also describe properties of the pattern itself, such as inverted matching.
- **Matching pattern**: traffic properties to look for, a combination of any number of items.
- **Actions** (represented by `struct rte_flow_action`): operations to perform whenever a packet is matched by a pattern.

Attributes

Attribute: Group

Flow rules can be grouped by assigning them a common group number. Groups allow a logical hierarchy of flow rule groups (tables) to be defined. These groups can be supported virtually in the PMD or in the physical device. Group 0 is the default group and this is the only group which flows are guarantee to matched against, all subsequent groups can only be reached by way of the JUMP action from a matched flow rule.

Although optional, applications are encouraged to group similar rules as much as possible to fully take advantage of hardware capabilities (e.g. optimized matching) and work around limitations (e.g. a single pattern type possibly allowed in a given group), while being aware that the groups hierarchies must be programmed explicitly.

Note that support for more than a single group is not guaranteed.

Attribute: Priority

A priority level can be assigned to a flow rule, lower values denote higher priority, with 0 as the maximum.

Priority levels are arbitrary and up to the application, they do not need to be contiguous nor start from 0, however the maximum number varies between devices and may be affected by existing flow rules.

A flow which matches multiple rules in the same group will always matched by the rule with the highest priority in that group.

If a packet is matched by several rules of a given group for a given priority level, the outcome is undefined. It can take any path, may be duplicated or even cause unrecoverable errors.

Note that support for more than a single priority level is not guaranteed.

Attribute: Traffic direction

Flow rule patterns apply to inbound and/or outbound traffic.

In the context of this API, **ingress** and **egress** respectively stand for **inbound** and **outbound** based on the standpoint of the application creating a flow rule.

There are no exceptions to this definition.

Several pattern items and actions are valid and can be used in both directions. At least one direction must be specified.

Specifying both directions at once for a given rule is not recommended but may be valid in a few cases (e.g. shared counters).

Attribute: Transfer

Instead of simply matching the properties of traffic as it would appear on a given DPDK port ID, enabling this attribute transfers a flow rule to the lowest possible level of any device endpoints found in the pattern.

When supported, this effectively enables an application to reroute traffic not necessarily intended for it (e.g. coming from or addressed to different physical ports, VFs or applications) at the device level.

It complements the behavior of some pattern items such as *Item: PHY_PORT* and is meaningless without them.

When transferring flow rules, **ingress** and **egress** attributes (*Attribute: Traffic direction*) keep their original meaning, as if processing traffic emitted or received by the application.

Pattern item

Pattern items fall in two categories:

- Matching protocol headers and packet data, usually associated with a specification structure. These must be stacked in the same order as the protocol layers to match inside packets, starting from the lowest.
- Matching meta-data or affecting pattern processing, often without a specification structure. Since they do not match packet contents, their position in the list is usually not relevant.

Item specification structures are used to match specific values among protocol fields (or item properties). Documentation describes for each item whether they are associated with one and their type name if so.

Up to three structures of the same type can be set for a given item:

- `spec`: values to match (e.g. a given IPv4 address).
- `last`: upper bound for an inclusive range with corresponding fields in `spec`.
- `mask`: bit-mask applied to both `spec` and `last` whose purpose is to distinguish the values to take into account and/or partially mask them out (e.g. in order to match an IPv4 address prefix).

Usage restrictions and expected behavior:

- Setting either `mask` or `last` without `spec` is an error.
- Field values in `last` which are either 0 or equal to the corresponding values in `spec` are ignored; they do not generate a range. Nonzero values lower than those in `spec` are not supported.
- Setting `spec` and optionally `last` without `mask` causes the PMD to use the default mask defined for that item (defined as `rte_flow_item_{name}_mask` constants).
- Not setting any of them (assuming item type allows it) is equivalent to providing an empty (zeroed) `mask` for broad (nonspecific) matching.
- `mask` is a simple bit-mask applied before interpreting the contents of `spec` and `last`, which may yield unexpected results if not used carefully. For example, if for an IPv4 address field, `spec` provides *10.1.2.3*, `last` provides *10.3.4.5* and `mask` provides *255.255.0.0*, the effective range becomes *10.1.0.0* to *10.3.255.255*.

Example of an item specification matching an Ethernet header:

Table 5.5: Ethernet item

Field	Subfield	Value
`spec`	`src`	`00:00:01:02:03:04`
	`dst`	`00:00:2a:66:00:01`
	`type`	`0x22aa`
`last`	unspecified	
`mask`	`src`	`00:00:ff:ff:ff:00`
	`dst`	`00:00:00:00:00:ff`
	`type`	`0x0000`

Non-masked bits stand for any value (shown as `?` below), Ethernet headers with the following properties are thus matched:

- `src`: `??:??:01:02:03:??`
- `dst`: `??:??:??:??:??:01`
- `type`: `0x????`

Matching pattern

A pattern is formed by stacking items starting from the lowest protocol layer to match. This stacking restriction does not apply to meta items which can be placed anywhere in the stack without affecting the meaning of the resulting pattern.

Patterns are terminated by END items.

Examples:

Table 5.6: TCPv4 as L4

Index	Item
0	Ethernet
1	IPv4
2	TCP
3	END

Table 5.7: TCPv6 in VXLAN

Index	Item
0	Ethernet
1	IPv4
2	UDP
3	VXLAN
4	Ethernet
5	IPv6
6	TCP
7	END

Table 5.8: TCPv4 as L4 with meta items

Index	Item
0	VOID
1	Ethernet
2	VOID
3	IPv4
4	TCP
5	VOID
6	VOID
7	END

The above example shows how meta items do not affect packet data matching items, as long as those remain stacked properly. The resulting matching pattern is identical to “TCPv4 as L4”.

Table 5.9: UDPv6 anywhere

Index	Item
0	IPv6
1	UDP
2	END

If supported by the PMD, omitting one or several protocol layers at the bottom of the stack as in the above example (missing an Ethernet specification) enables looking up anywhere in packets.

It is unspecified whether the payload of supported encapsulations (e.g. VXLAN payload) is matched by such a pattern, which may apply to inner, outer or both packets.

Table 5.10: Invalid, missing L3

Index	Item
0	Ethernet
1	UDP
2	END

The above pattern is invalid due to a missing L3 specification between L2 (Ethernet) and L4 (UDP). Doing so is only allowed at the bottom and at the top of the stack.

Meta item types

They match meta-data or affect pattern processing instead of matching packet data directly, most of them do not need a specification structure. This particularity allows them to be specified anywhere in the stack without causing any side effect.

Item: `END`

End marker for item lists. Prevents further processing of items, thereby ending the pattern.

- Its numeric value is 0 for convenience.
- PMD support is mandatory.
- `spec`, `last` and `mask` are ignored.

Table 5.11: END

Field	Value
`spec`	ignored
`last`	ignored
`mask`	ignored

Item: VOID

Used as a placeholder for convenience. It is ignored and simply discarded by PMDs.

- PMD support is mandatory.
- spec, last and mask are ignored.

Table 5.12: VOID

Field	Value
spec	ignored
last	ignored
mask	ignored

One usage example for this type is generating rules that share a common prefix quickly without reallocating memory, only by updating item types:

Table 5.13: TCP, UDP or ICMP as L4

Index	Item		
0	Ethernet		
1	IPv4		
2	UDP	VOID	VOID
3	VOID	TCP	VOID
4	VOID	VOID	ICMP
5	END		

Item: INVERT

Inverted matching, i.e. process packets that do not match the pattern.

- spec, last and mask are ignored.

Table 5.14: INVERT

Field	Value
spec	ignored
last	ignored
mask	ignored

Usage example, matching non-TCPv4 packets only:

Table 5.15: Anything but TCPv4

Index	Item
0	INVERT
1	Ethernet
2	IPv4
3	TCP
4	END

Item: PF

Matches traffic originating from (ingress) or going to (egress) the physical function of the current device.

If supported, should work even if the physical function is not managed by the application and thus not associated with a DPDK port ID.

- Can be combined with any number of *Item: VF* to match both PF and VF traffic.
- `spec`, `last` and `mask` must not be set.

Table 5.16: PF

Field	Value
`spec`	unset
`last`	unset
`mask`	unset

Item: VF

Matches traffic originating from (ingress) or going to (egress) a given virtual function of the current device.

If supported, should work even if the virtual function is not managed by the application and thus not associated with a DPDK port ID.

Note this pattern item does not match VF representors traffic which, as separate entities, should be addressed through their own DPDK port IDs.

- Can be specified multiple times to match traffic addressed to several VF IDs.
- Can be combined with a PF item to match both PF and VF traffic.
- Default `mask` matches any VF ID.

Table 5.17: VF

Field	Subfield	Value
`spec`	`id`	destination VF ID
`last`	`id`	upper range value
`mask`	`id`	zeroed to match any VF ID

Item: PHY_PORT

Matches traffic originating from (ingress) or going to (egress) a physical port of the underlying device.

The first PHY_PORT item overrides the physical port normally associated with the specified DPDK input port (port_id). This item can be provided several times to match additional physical ports.

Note that physical ports are not necessarily tied to DPDK input ports (port_id) when those are not under DPDK control. Possible values are specific to each device, they are not necessarily indexed from zero and may not be contiguous.

As a device property, the list of allowed values as well as the value associated with a port_id should be retrieved by other means.

- Default `mask` matches any port index.

Table 5.18: PHY_PORT

Field	Subfield	Value
`spec`	`index`	physical port index
`last`	`index`	upper range value
`mask`	`index`	zeroed to match any port index

Item: `PORT_ID`

Matches traffic originating from (ingress) or going to (egress) a given DPDK port ID.

Normally only supported if the port ID in question is known by the underlying PMD and related to the device the flow rule is created against.

This must not be confused with *Item: PHY_PORT* which refers to the physical port of a device, whereas *Item: PORT_ID* refers to a `struct rte_eth_dev` object on the application side (also known as "port representor" depending on the kind of underlying device).

- Default `mask` matches the specified DPDK port ID.

Table 5.19: PORT_ID

Field	Subfield	Value
`spec`	`id`	DPDK port ID
`last`	`id`	upper range value
`mask`	`id`	zeroed to match any port ID

Item: `MARK`

Matches an arbitrary integer value which was set using the `MARK` action in a previously matched rule.

This item can only specified once as a match criteria as the `MARK` action can only be specified once in a flow action.

Note the value of MARK field is arbitrary and application defined.

Depending on the underlying implementation the MARK item may be supported on the physical device, with virtual groups in the PMD or not at all.

- Default `mask` matches any integer value.

Table 5.20: MARK

Field	Subfield	Value
`spec`	`id` | integer value	
`last`	`id` | upper range value	
`mask`	`id`	zeroed to match any value

Item: TAG

Matches tag item set by other flows. Multiple tags are supported by specifying `index`.

- Default `mask` matches the specified tag value and index.

Table 5.21: TAG

Field	Subfield | Value	
`spec`	`data`	32 bit flow tag value
	`index`	index of flow tag
`last`	`data`	upper range value
	`index`	field is ignored
`mask`	`data`	bit-mask applies to "spec" and "last"
	`index`	field is ignored

Item: META

Matches 32 bit metadata item set.

On egress, metadata can be set either by mbuf metadata field with PKT_TX_DYNF_METADATA flag or `SET_META` action. On ingress, `SET_META` action sets metadata for a packet and the metadata will be reported via `metadata` dynamic field of `rte_mbuf` with PKT_RX_DYNF_METADATA flag.

- Default `mask` matches the specified Rx metadata value.

Table 5.22: META

Field	Subfield	Value
`spec`	`data`	32 bit metadata value
`last`	`data`	upper range value
`mask`	`data`	bit-mask applies to "spec" and "last"

Data matching item types

Most of these are basically protocol header definitions with associated bit-masks. They must be specified (stacked) from lowest to highest protocol layer to form a matching pattern.

The following list is not exhaustive, new protocols will be added in the future.

Item: ANY

Matches any protocol in place of the current layer, a single ANY may also stand for several protocol layers.

This is usually specified as the first pattern item when looking for a protocol anywhere in a packet.

- Default `mask` stands for any number of layers.

Table 5.23: ANY

Field	Subfield	Value
spec	num	number of layers covered
last	num	upper range value
mask	num	zeroed to cover any number of layers

Example for VXLAN TCP payload matching regardless of outer L3 (IPv4 or IPv6) and L4 (UDP) both matched by the first ANY specification, and inner L3 (IPv4 or IPv6) matched by the second ANY specification:

Table 5.24: TCP in VXLAN with wildcards

Index	Item	Field	Subfield	Value
0	Ethernet			
1	ANY	spec	num	2
2	VXLAN			
3	Ethernet			
4	ANY	spec	num	1
5	TCP			
6	END			

Item: RAW

Matches a byte string of a given length at a given offset.

Offset is either absolute (using the start of the packet) or relative to the end of the previous matched item in the stack, in which case negative values are allowed.

If search is enabled, offset is used as the starting point. The search area can be delimited by setting limit to a nonzero value, which is the maximum number of bytes after offset where the pattern may start.

Matching a zero-length pattern is allowed, doing so resets the relative offset for subsequent items.

- This type does not support ranges (last field).
- Default mask matches all fields exactly.

Table 5.25: RAW

Field	Subfield	Value
spec	relative	look for pattern after the previous item
	search	search pattern from offset (see also limit)
	reserved	reserved, must be set to zero
	offset	absolute or relative offset for pattern
	limit	search area limit for start of pattern
	length	pattern length
	pattern	byte string to look for
last	if specified, either all 0 or with the same values as spec	
mask	bit-mask applied to spec values with usual behavior	

Example pattern looking for several strings at various offsets of a UDP payload, using combined RAW items:

Table 5.26: UDP payload matching

Index	Item	Field	Subfield	Value
0	Ethernet			
1	IPv4			
2	UDP			
3	RAW	spec	relative	1
			search	1
			offset	10
			limit	0
			length	3
			pattern	“foo”
4	RAW	spec	relative	1
			search	0
			offset	20
			limit	0
			length	3
			pattern	“bar”
5	RAW	spec	relative	1
			search	0
			offset	-29
			limit	0
			length	3
			pattern	“baz”
6	END			

This translates to:

- Locate “foo” at least 10 bytes deep inside UDP payload.
- Locate “bar” after “foo” plus 20 bytes.
- Locate “baz” after “bar” minus 29 bytes.

Such a packet may be represented as follows (not to scale):

```
0                     >= 10 B           == 20 B
|                   |<---------->|     |<---------->|
|                   |            |     |            |
|-----|------|-----|-----|-----|-----|-----------|-----|------|
| ETH | IPv4 | UDP | ... | baz | foo | ......... | bar | .... |
|-----|------|-----|-----|-----|-----|-----------|-----|------|
                        |                           |
                        |<------------------------->|
                                   == 29 B
```

Note that matching subsequent pattern items would resume after “baz”, not “bar” since matching is always performed after the previous item of the stack.

Item: ETH

Matches an Ethernet header.

The type field either stands for "EtherType" or "TPID" when followed by so-called layer 2.5 pattern items such as RTE_FLOW_ITEM_TYPE_VLAN. In the latter case, type refers to that of the outer header, with the inner EtherType/TPID provided by the subsequent pattern item. This is the same order as on the wire. If the type field contains a TPID value, then only tagged packets with the specified TPID will match the pattern. Otherwise, only untagged packets will match the pattern. If the ETH item is the only item in the pattern, and the type field is not specified, then both tagged and untagged packets will match the pattern.

- dst: destination MAC.
- src: source MAC.
- type: EtherType or TPID.
- Default mask matches destination and source addresses only.

Item: VLAN

Matches an 802.1Q/ad VLAN tag.

The corresponding standard outer EtherType (TPID) values are RTE_ETHER_TYPE_VLAN or RTE_ETHER_TYPE_QINQ. It can be overridden by the preceding pattern item. If a VLAN item is present in the pattern, then only tagged packets will match the pattern.

- tci: tag control information.
- inner_type: inner EtherType or TPID.
- Default mask matches the VID part of TCI only (lower 12 bits).

Item: IPV4

Matches an IPv4 header.

Note: IPv4 options are handled by dedicated pattern items.

- hdr: IPv4 header definition (rte_ip.h).
- Default mask matches source and destination addresses only.

Item: IPV6

Matches an IPv6 header.

Note: IPv6 options are handled by dedicated pattern items, see *Item: IPV6_EXT*.

- hdr: IPv6 header definition (rte_ip.h).
- Default mask matches source and destination addresses only.

Item: ICMP

Matches an ICMP header.

- hdr: ICMP header definition (rte_icmp.h).
- Default mask matches ICMP type and code only.

Item: UDP

Matches a UDP header.

- hdr: UDP header definition (rte_udp.h).
- Default mask matches source and destination ports only.

Item: TCP

Matches a TCP header.

- hdr: TCP header definition (rte_tcp.h).
- Default mask matches source and destination ports only.

Item: SCTP

Matches a SCTP header.

- hdr: SCTP header definition (rte_sctp.h).
- Default mask matches source and destination ports only.

Item: VXLAN

Matches a VXLAN header (RFC 7348).

- flags: normally 0x08 (I flag).
- rsvd0: reserved, normally 0x000000.
- vni: VXLAN network identifier.
- rsvd1: reserved, normally 0x00.
- Default mask matches VNI only.

Item: E_TAG

Matches an IEEE 802.1BR E-Tag header.

The corresponding standard outer EtherType (TPID) value is RTE_ETHER_TYPE_ETAG. It can be overridden by the preceding pattern item.

- epcp_edei_in_ecid_b: E-Tag control information (E-TCI), E-PCP (3b), E-DEI (1b), ingress E-CID base (12b).
- rsvd_grp_ecid_b: reserved (2b), GRP (2b), E-CID base (12b).
- in_ecid_e: ingress E-CID ext.
- ecid_e: E-CID ext.
- inner_type: inner EtherType or TPID.
- Default mask simultaneously matches GRP and E-CID base.

Item: NVGRE

Matches a NVGRE header (RFC 7637).

- c_k_s_rsvd0_ver: checksum (1b), undefined (1b), key bit (1b), sequence number (1b), reserved 0 (9b), version (3b). This field must have value 0x2000 according to RFC 7637.
- protocol: protocol type (0x6558).
- tni: virtual subnet ID.
- flow_id: flow ID.
- Default mask matches TNI only.

Item: MPLS

Matches a MPLS header.

- label_tc_s_ttl: label, TC, Bottom of Stack and TTL.
- Default mask matches label only.

Item: GRE

Matches a GRE header.

- c_rsvd0_ver: checksum, reserved 0 and version.
- protocol: protocol type.
- Default mask matches protocol only.

Item: GRE_KEY

Matches a GRE key field. This should be preceded by item GRE.

- Value to be matched is a big-endian 32 bit integer.
- When this item present it implicitly match K bit in default mask as "1"

Item: FUZZY

Fuzzy pattern match, expect faster than default.

This is for device that support fuzzy match option. Usually a fuzzy match is fast but the cost is accuracy. i.e. Signature Match only match pattern's hash value, but it is possible two different patterns have the same hash value.

Matching accuracy level can be configured by threshold. Driver can divide the range of threshold and map to different accuracy levels that device support.

Threshold 0 means perfect match (no fuzziness), while threshold 0xffffffff means fuzziest match.

Table 5.27: FUZZY

Field	Subfield	Value
spec	threshold	0 as perfect match, 0xffffffff as fuzziest match
last	threshold	upper range value
mask	threshold	bit-mask apply to "spec" and "last"

Usage example, fuzzy match a TCPv4 packets:

Table 5.28: Fuzzy matching

Index	Item
0	FUZZY
1	Ethernet
2	IPv4
3	TCP
4	END

Item: GTP, GTPC, GTPU

Matches a GTPv1 header.

Note: GTP, GTPC and GTPU use the same structure. GTPC and GTPU item are defined for a user-friendly API when creating GTP-C and GTP-U flow rules.

- v_pt_rsv_flags: version (3b), protocol type (1b), reserved (1b), extension header flag (1b), sequence number flag (1b), N-PDU number flag (1b).
- msg_type: message type.
- msg_len: message length.
- teid: tunnel endpoint identifier.

- Default `mask` matches teid only.

Item: `ESP`

Matches an ESP header.

- `hdr`: ESP header definition (`rte_esp.h`).
- Default `mask` matches SPI only.

Item: `GENEVE`

Matches a GENEVE header.

- `ver_opt_len_o_c_rsvd0`: version (2b), length of the options fields (6b), OAM packet (1b), critical options present (1b), reserved 0 (6b).
- `protocol`: protocol type.
- `vni`: virtual network identifier.
- `rsvd1`: reserved, normally 0x00.
- Default `mask` matches VNI only.

Item: `VXLAN-GPE`

Matches a VXLAN-GPE header (draft-ietf-nvo3-vxlan-gpe-05).

- `flags`: normally 0x0C (I and P flags).
- `rsvd0`: reserved, normally 0x0000.
- `protocol`: protocol type.
- `vni`: VXLAN network identifier.
- `rsvd1`: reserved, normally 0x00.
- Default `mask` matches VNI only.

Item: `ARP_ETH_IPV4`

Matches an ARP header for Ethernet/IPv4.

- `hdr`: hardware type, normally 1.
- `pro`: protocol type, normally 0x0800.
- `hln`: hardware address length, normally 6.
- `pln`: protocol address length, normally 4.
- `op`: opcode (1 for request, 2 for reply).
- `sha`: sender hardware address.
- `spa`: sender IPv4 address.

- `tha`: target hardware address.
- `tpa`: target IPv4 address.
- Default `mask` matches SHA, SPA, THA and TPA.

Item: `IPV6_EXT`

Matches the presence of any IPv6 extension header.

- `next_hdr`: next header.
- Default `mask` matches `next_hdr`.

Normally preceded by any of:

- *Item: IPV6*
- *Item: IPV6_EXT*

Item: `ICMP6`

Matches any ICMPv6 header.

- `type`: ICMPv6 type.
- `code`: ICMPv6 code.
- `checksum`: ICMPv6 checksum.
- Default `mask` matches `type` and `code`.

Item: `ICMP6_ND_NS`

Matches an ICMPv6 neighbor discovery solicitation.

- `type`: ICMPv6 type, normally 135.
- `code`: ICMPv6 code, normally 0.
- `checksum`: ICMPv6 checksum.
- `reserved`: reserved, normally 0.
- `target_addr`: target address.
- Default `mask` matches target address only.

Item: ICMP6_ND_NA

Matches an ICMPv6 neighbor discovery advertisement.

- `type`: ICMPv6 type, normally 136.
- `code`: ICMPv6 code, normally 0.
- `checksum`: ICMPv6 checksum.
- `rso_reserved`: route flag (1b), solicited flag (1b), override flag (1b), reserved (29b).
- `target_addr`: target address.
- Default `mask` matches target address only.

Item: ICMP6_ND_OPT

Matches the presence of any ICMPv6 neighbor discovery option.

- `type`: ND option type.
- `length`: ND option length.
- Default `mask` matches type only.

Normally preceded by any of:

- *Item: ICMP6_ND_NA*
- *Item: ICMP6_ND_NS*
- *Item: ICMP6_ND_OPT*

Item: ICMP6_ND_OPT_SLA_ETH

Matches an ICMPv6 neighbor discovery source Ethernet link-layer address option.

- `type`: ND option type, normally 1.
- `length`: ND option length, normally 1.
- `sla`: source Ethernet LLA.
- Default `mask` matches source link-layer address only.

Normally preceded by any of:

- *Item: ICMP6_ND_NA*
- *Item: ICMP6_ND_OPT*

Item: `ICMP6_ND_OPT_TLA_ETH`

Matches an ICMPv6 neighbor discovery target Ethernet link-layer address option.

- `type`: ND option type, normally 2.
- `length`: ND option length, normally 1.
- `tla`: target Ethernet LLA.
- Default `mask` matches target link-layer address only.

Normally preceded by any of:

- *Item: ICMP6_ND_NS*
- *Item: ICMP6_ND_OPT*

Item: `META`

Matches an application specific 32 bit metadata item.

- Default `mask` matches the specified metadata value.

Item: `GTP_PSC`

Matches a GTP PDU extension header with type 0x85.

- `pdu_type`: PDU type.
- `qfi`: QoS flow identifier.
- Default `mask` matches QFI only.

Item: `PPPOES, PPPOED`

Matches a PPPoE header.

- `version_type`: version (4b), type (4b).
- `code`: message type.
- `session_id`: session identifier.
- `length`: payload length.

Item: PPPOE_PROTO_ID

Matches a PPPoE session protocol identifier.

- proto_id: PPP protocol identifier.
- Default mask matches proto_id only.

Item: NSH

Matches a network service header (RFC 8300).

- version: normally 0x0 (2 bits).
- oam_pkt: indicate oam packet (1 bit).
- reserved: reserved bit (1 bit).
- ttl: maximum SFF hopes (6 bits).
- length: total length in 4 bytes words (6 bits).
- reserved1: reserved1 bits (4 bits).
- mdtype: ndicates format of NSH header (4 bits).
- next_proto: indicates protocol type of encap data (8 bits).
- spi: service path identifier (3 bytes).
- sindex: service index (1 byte).
- Default mask matches mdtype, next_proto, spi, sindex.

Item: IGMP

Matches a Internet Group Management Protocol (RFC 2236).

- type: IGMP message type (Query/Report).
- max_resp_time: max time allowed before sending report.
- checksum: checksum, 1s complement of whole IGMP message.
- group_addr: group address, for Query value will be 0.
- Default mask matches group_addr.

Item: AH

Matches a IP Authentication Header (RFC 4302).

- next_hdr: next payload after AH.
- payload_len: total length of AH in 4B words.
- reserved: reserved bits.
- spi: security parameters index.

- `seq_num`: counter value increased by 1 on each packet sent.
- Default `mask` matches spi.

Item: `HIGIG2`

Matches a HIGIG2 header field. It is layer 2.5 protocol and used in Broadcom switches.

- Default `mask` matches classification and vlan.

Item: `L2TPV3OIP`

Matches a L2TPv3 over IP header.

- `session_id`: L2TPv3 over IP session identifier.
- Default `mask` matches session_id only.

Item: `PFCP`

Matches a PFCP Header.

- `s_field`: S field.
- `msg_type`: message type.
- `msg_len`: message length.
- `seid`: session endpoint identifier.
- Default `mask` matches s_field and seid.

Actions

Each possible action is represented by a type. An action can have an associated configuration object. Several actions combined in a list can be assigned to a flow rule and are performed in order.

They fall in three categories:

- Actions that modify the fate of matching traffic, for instance by dropping or assigning it a specific destination.
- Actions that modify matching traffic contents or its properties. This includes adding/removing encapsulation, encryption, compression and marks.
- Actions related to the flow rule itself, such as updating counters or making it non-terminating.

Flow rules being terminating by default, not specifying any action of the fate kind results in undefined behavior. This applies to both ingress and egress.

PASSTHRU, when supported, makes a flow rule non-terminating.

Like matching patterns, action lists are terminated by END items.

Example of action that redirects packets to queue index 10:

Table 5.29: Queue action

Field	Value
index	10

Actions are performed in list order:

Table 5.30: Count then drop

Index	Action
0	COUNT
1	DROP
2	END

Table 5.31: Mark, count then redirect

Index	Action	Field	Value
0	MARK	mark	0x2a
1	COUNT	shared	0
		id	0
2	QUEUE	queue	10
3	END		

Table 5.32: Redirect to queue 5

Index	Action	Field	Value
0	DROP		
1	QUEUE	queue	5
2	END		

In the above example, while DROP and QUEUE must be performed in order, both have to happen before reaching END. Only QUEUE has a visible effect.

Note that such a list may be thought as ambiguous and rejected on that basis.

Table 5.33: Redirect to queues 5 and 3

Index	Action	Field	Value
0	QUEUE	queue	5
1	VOID		
2	QUEUE	queue	3
3	END		

As previously described, all actions must be taken into account. This effectively duplicates traffic to both queues. The above example also shows that VOID is ignored.

Action types

Common action types are described in this section. Like pattern item types, this list is not exhaustive as new actions will be added in the future.

Action: `END`

End marker for action lists. Prevents further processing of actions, thereby ending the list.

- Its numeric value is 0 for convenience.
- PMD support is mandatory.
- No configurable properties.

Table 5.34: END

Field
no properties

Action: `VOID`

Used as a placeholder for convenience. It is ignored and simply discarded by PMDs.

- PMD support is mandatory.
- No configurable properties.

Table 5.35: VOID

Field
no properties

Action: `PASSTHRU`

Leaves traffic up for additional processing by subsequent flow rules; makes a flow rule non-terminating.

- No configurable properties.

Table 5.36: PASSTHRU

Field
no properties

Example to copy a packet to a queue and continue processing by subsequent flow rules:

Table 5.37: Copy to queue 8

Index	Action	Field	Value
0	PASSTHRU		
1	QUEUE	queue	8
2	END		

Action: JUMP

Redirects packets to a group on the current device.

In a hierarchy of groups, which can be used to represent physical or logical flow group/tables on the device, this action redirects the matched flow to the specified group on that device.

If a matched flow is redirected to a table which doesn't contain a matching rule for that flow then the behavior is undefined and the resulting behavior is up to the specific device. Best practice when using groups would be define a default flow rule for each group which a defines the default actions in that group so a consistent behavior is defined.

Defining an action for matched flow in a group to jump to a group which is higher in the group hierarchy may not be supported by physical devices, depending on how groups are mapped to the physical devices. In the definitions of jump actions, applications should be aware that it may be possible to define flow rules which trigger an undefined behavior causing flows to loop between groups.

Table 5.38: JUMP

Field	Value
group	Group to redirect packets to

Action: MARK

Attaches an integer value to packets and sets PKT_RX_FDIR and PKT_RX_FDIR_ID mbuf flags.

This value is arbitrary and application-defined. Maximum allowed value depends on the underlying implementation. It is returned in the hash.fdir.hi mbuf field.

Table 5.39: MARK

Field	Value
id	integer value to return with packets

Action: FLAG

Flags packets. Similar to *Action: MARK* without a specific value; only sets the PKT_RX_FDIR mbuf flag.

- No configurable properties.

Table 5.40: FLAG

Field
no properties

Action: `QUEUE`

Assigns packets to a given queue index.

Table 5.41: QUEUE

Field	Value
`index`	queue index to use

Action: `DROP`

Drop packets.

- No configurable properties.

Table 5.42: DROP

Field
no properties

Action: `COUNT`

Adds a counter action to a matched flow.

If more than one count action is specified in a single flow rule, then each action must specify a unique id.

Counters can be retrieved and reset through `rte_flow_query()`, see `struct rte_flow_query_count`.

The shared flag indicates whether the counter is unique to the flow rule the action is specified with, or whether it is a shared counter.

For a count action with the shared flag set, then a global device namespace is assumed for the counter id, so that any matched flow rules using a count action with the same counter id on the same port will contribute to that counter.

For ports within the same switch domain then the counter id namespace extends to all ports within that switch domain.

Table 5.43: COUNT

Field	Value
`shared`	shared counter flag
`id`	counter id

Query structure to retrieve and reset flow rule counters:

Table 5.44: COUNT query

Field	I/O	Value
`reset`	in	reset counter after query
`hits_set`	out	`hits` field is set
`bytes_set`	out	`bytes` field is set
`hits`	out	number of hits for this rule
`bytes`	out	number of bytes through this rule

Action: RSS

Similar to QUEUE, except RSS is additionally performed on packets to spread them among several queues according to the provided parameters.

Unlike global RSS settings used by other DPDK APIs, unsetting the `types` field does not disable RSS in a flow rule. Doing so instead requests safe unspecified "best-effort" settings from the underlying PMD, which depending on the flow rule, may result in anything ranging from empty (single queue) to all-inclusive RSS.

Note: RSS hash result is stored in the `hash.rss` mbuf field which overlaps `hash.fdir.lo`. Since *Action: MARK* sets the `hash.fdir.hi` field only, both can be requested simultaneously.

Also, regarding packet encapsulation `level`:

- `0` requests the default behavior. Depending on the packet type, it can mean outermost, innermost, anything in between or even no RSS.

  It basically stands for the innermost encapsulation level RSS can be performed on according to PMD and device capabilities.

- `1` requests RSS to be performed on the outermost packet encapsulation level.

- **`2` and subsequent values request RSS to be performed on the specified**
  inner packet encapsulation level, from outermost to innermost (lower to higher values).

Values other than `0` are not necessarily supported.

Requesting a specific RSS level on unrecognized traffic results in undefined behavior. For predictable results, it is recommended to make the flow rule pattern match packet headers up to the requested encapsulation level so that only matching traffic goes through.

Table 5.45: RSS

Field	Value
`func`	RSS hash function to apply
`level`	encapsulation level for `types`
`types`	specific RSS hash types (see `ETH_RSS_*`)
`key_len`	hash key length in bytes
`queue_num`	number of entries in `queue`
`key`	hash key
`queue`	queue indices to use

Action: `PF`

Directs matching traffic to the physical function (PF) of the current device.

See *Item: PF*.

- No configurable properties.

Table 5.46: PF

Field
no properties

Action: `VF`

Directs matching traffic to a given virtual function of the current device.

Packets matched by a VF pattern item can be redirected to their original VF ID instead of the specified one. This parameter may not be available and is not guaranteed to work properly if the VF part is matched by a prior flow rule or if packets are not addressed to a VF in the first place.

See *Item: VF*.

Table 5.47: VF

Field	Value
`original`	use original VF ID if possible
`id`	VF ID

Action: `PHY_PORT`

Directs matching traffic to a given physical port index of the underlying device.

See *Item: PHY_PORT*.

Table 5.48: PHY_PORT

Field	Value
`original`	use original port index if possible
`index`	physical port index

Action: `PORT_ID`

Directs matching traffic to a given DPDK port ID.

See *Item: PORT_ID*.

Table 5.49: PORT_ID

Field	Value
`original`	use original DPDK port ID if possible
`id`	DPDK port ID

Action: `METER`

Applies a stage of metering and policing.

The metering and policing (MTR) object has to be first created using the rte_mtr_create() API function. The ID of the MTR object is specified as action parameter. More than one flow can use the same MTR object through the meter action. The MTR object can be further updated or queried using the rte_mtr* API.

Table 5.50: METER

Field	Value
`mtr_id`	MTR object ID

Action: `SECURITY`

Perform the security action on flows matched by the pattern items according to the configuration of the security session.

This action modifies the payload of matched flows. For INLINE_CRYPTO, the security protocol headers and IV are fully provided by the application as specified in the flow pattern. The payload of matching packets is encrypted on egress, and decrypted and authenticated on ingress. For INLINE_PROTOCOL, the security protocol is fully offloaded to HW, providing full encapsulation and decapsulation of packets in security protocols. The flow pattern specifies both the outer security header fields and the inner packet fields. The security session specified in the action must match the pattern parameters.

The security session specified in the action must be created on the same port as the flow action that is being specified.

The ingress/egress flow attribute should match that specified in the security session if the security session supports the definition of the direction.

Multiple flows can be configured to use the same security session.

Table 5.51: SECURITY

Field	Value
`security_session`	security session to apply

The following is an example of configuring IPsec inline using the INLINE_CRYPTO security session:

The encryption algorithm, keys and salt are part of the opaque `rte_security_session`. The SA is identified according to the IP and ESP fields in the pattern items.

Table 5.52: IPsec inline crypto flow pattern items.

Index	Item
0	Ethernet
1	IPv4
2	ESP
3	END

Table 5.53: IPsec inline flow actions.

Index	Action
0	SECURITY
1	END

Action: `OF_SET_MPLS_TTL`

Implements `OFPAT_SET_MPLS_TTL` ("MPLS TTL") as defined by the OpenFlow Switch Specification.

Table 5.54: OF_SET_MPLS_TTL

Field	Value
`mpls_ttl`	MPLS TTL

Action: `OF_DEC_MPLS_TTL`

Implements `OFPAT_DEC_MPLS_TTL` ("decrement MPLS TTL") as defined by the OpenFlow Switch Specification.

Table 5.55: OF_DEC_MPLS_TTL

Field
no properties

Action: `OF_SET_NW_TTL`

Implements `OFPAT_SET_NW_TTL` ("IP TTL") as defined by the OpenFlow Switch Specification.

Table 5.56: OF_SET_NW_TTL

Field	Value
`nw_ttl`	IP TTL

Action: `OF_DEC_NW_TTL`

Implements `OFPAT_DEC_NW_TTL` ("decrement IP TTL") as defined by the OpenFlow Switch Specification.

Table 5.57: OF_DEC_NW_TTL

Field
no properties

Action: `OF_COPY_TTL_OUT`

Implements `OFPAT_COPY_TTL_OUT` ("copy TTL "outwards" – from next-to-outermost to outermost") as defined by the OpenFlow Switch Specification.

Table 5.58: OF_COPY_TTL_OUT

Field
no properties

Action: `OF_COPY_TTL_IN`

Implements `OFPAT_COPY_TTL_IN` ("copy TTL "inwards" – from outermost to next-to-outermost") as defined by the OpenFlow Switch Specification.

Table 5.59: OF_COPY_TTL_IN

Field
no properties

Action: `OF_POP_VLAN`

Implements `OFPAT_POP_VLAN` ("pop the outer VLAN tag") as defined by the OpenFlow Switch Specification.

Table 5.60: OF_POP_VLAN

Field
no properties

Action: `OF_PUSH_VLAN`

Implements `OFPAT_PUSH_VLAN` ("push a new VLAN tag") as defined by the OpenFlow Switch Specification.

Table 5.61: OF_PUSH_VLAN

Field	Value
`ethertype`	EtherType

Action: OF_SET_VLAN_VID

Implements OFPAT_SET_VLAN_VID ("set the 802.1q VLAN id") as defined by the OpenFlow Switch Specification.

Table 5.62: OF_SET_VLAN_VID

Field	Value
vlan_vid	VLAN id

Action: OF_SET_VLAN_PCP

Implements OFPAT_SET_LAN_PCP ("set the 802.1q priority") as defined by the OpenFlow Switch Specification.

Table 5.63: OF_SET_VLAN_PCP

Field	Value
vlan_pcp	VLAN priority

Action: OF_POP_MPLS

Implements OFPAT_POP_MPLS ("pop the outer MPLS tag") as defined by the OpenFlow Switch Specification.

Table 5.64: OF_POP_MPLS

Field	Value
ethertype	EtherType

Action: OF_PUSH_MPLS

Implements OFPAT_PUSH_MPLS ("push a new MPLS tag") as defined by the OpenFlow Switch Specification.

Table 5.65: OF_PUSH_MPLS

Field	Value
ethertype	EtherType

Action: VXLAN_ENCAP

Performs a VXLAN encapsulation action by encapsulating the matched flow in the VXLAN tunnel as defined in the``rte_flow_action_vxlan_encap`` flow items definition.

This action modifies the payload of matched flows. The flow definition specified in the `rte_flow_action_tunnel_encap` action structure must define a valid VLXAN network overlay which conforms with RFC 7348 (Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks). The pattern must be terminated with the RTE_FLOW_ITEM_TYPE_END item type.

Table 5.66: VXLAN_ENCAP

Field	Value
`definition`	Tunnel end-point overlay definition

Table 5.67: IPv4 VxLAN flow pattern example.

Index	Item
0	Ethernet
1	IPv4
2	UDP
3	VXLAN
4	END

Action: VXLAN_DECAP

Performs a decapsulation action by stripping all headers of the VXLAN tunnel network overlay from the matched flow.

The flow items pattern defined for the flow rule with which a `VXLAN_DECAP` action is specified, must define a valid VXLAN tunnel as per RFC7348. If the flow pattern does not specify a valid VXLAN tunnel then a RTE_FLOW_ERROR_TYPE_ACTION error should be returned.

This action modifies the payload of matched flows.

Action: NVGRE_ENCAP

Performs a NVGRE encapsulation action by encapsulating the matched flow in the NVGRE tunnel as defined in the``rte_flow_action_tunnel_encap`` flow item definition.

This action modifies the payload of matched flows. The flow definition specified in the `rte_flow_action_tunnel_encap` action structure must defined a valid NVGRE network overlay which conforms with RFC 7637 (NVGRE: Network Virtualization Using Generic Routing Encapsulation). The pattern must be terminated with the RTE_FLOW_ITEM_TYPE_END item type.

Table 5.68: NVGRE_ENCAP

Field	Value
`definition`	NVGRE end-point overlay definition

Table 5.69: IPv4 NVGRE flow pattern example.

Index	Item
0	Ethernet
1	IPv4
2	NVGRE
3	END

Action: NVGRE_DECAP

Performs a decapsulation action by stripping all headers of the NVGRE tunnel network overlay from the matched flow.

The flow items pattern defined for the flow rule with which a NVGRE_DECAP action is specified, must define a valid NVGRE tunnel as per RFC7637. If the flow pattern does not specify a valid NVGRE tunnel then a RTE_FLOW_ERROR_TYPE_ACTION error should be returned.

This action modifies the payload of matched flows.

Action: RAW_ENCAP

Adds outer header whose template is provided in its data buffer, as defined in the rte_flow_action_raw_encap definition.

This action modifies the payload of matched flows. The data supplied must be a valid header, either holding layer 2 data in case of adding layer 2 after decap layer 3 tunnel (for example MPLSoGRE) or complete tunnel definition starting from layer 2 and moving to the tunnel item itself. When applied to the original packet the resulting packet must be a valid packet.

Table 5.70: RAW_ENCAP

Field	Value
data	Encapsulation data
preserve	Bit-mask of data to preserve on output
size	Size of data and preserve

Action: RAW_DECAP

Remove outer header whose template is provided in its data buffer, as defined in the rte_flow_action_raw_decap

This action modifies the payload of matched flows. The data supplied must be a valid header, either holding layer 2 data in case of removing layer 2 before encapsulation of layer 3 tunnel (for example MPLSoGRE) or complete tunnel definition starting from layer 2 and moving to the tunnel item itself. When applied to the original packet the resulting packet must be a valid packet.

Table 5.71: RAW_DECAP

Field	Value
data	Decapsulation data
size	Size of data

Action: SET_IPV4_SRC

Set a new IPv4 source address in the outermost IPv4 header.

It must be used with a valid RTE_FLOW_ITEM_TYPE_IPV4 flow pattern item. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.72: SET_IPV4_SRC

Field | Value	
ipv4_addr	new IPv4 source address

Action: SET_IPV4_DST

Set a new IPv4 destination address in the outermost IPv4 header.

It must be used with a valid RTE_FLOW_ITEM_TYPE_IPV4 flow pattern item. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.73: SET_IPV4_DST

Field	Value
ipv4_addr	new IPv4 destination address

Action: SET_IPV6_SRC

Set a new IPv6 source address in the outermost IPv6 header.

It must be used with a valid RTE_FLOW_ITEM_TYPE_IPV6 flow pattern item. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.74: SET_IPV6_SRC

Field	Value
ipv6_addr	new IPv6 source address

Action: SET_IPV6_DST

Set a new IPv6 destination address in the outermost IPv6 header.

It must be used with a valid RTE_FLOW_ITEM_TYPE_IPV6 flow pattern item. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.75: SET_IPV6_DST

Field	Value
ipv6_addr	new IPv6 destination address

Action: SET_TP_SRC

Set a new source port number in the outermost TCP/UDP header.

It must be used with a valid RTE_FLOW_ITEM_TYPE_TCP or RTE_FLOW_ITEM_TYPE_UDP flow pattern item. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.76: SET_TP_SRC

Field	Value
`port`	new TCP/UDP source port

Action: SET_TP_DST

Set a new destination port number in the outermost TCP/UDP header.

It must be used with a valid RTE_FLOW_ITEM_TYPE_TCP or RTE_FLOW_ITEM_TYPE_UDP flow pattern item. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.77: SET_TP_DST

Field	Value
`port`	new TCP/UDP destination port

Action: MAC_SWAP

Swap the source and destination MAC addresses in the outermost Ethernet header.

It must be used with a valid RTE_FLOW_ITEM_TYPE_ETH flow pattern item. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.78: MAC_SWAP

Field
no properties

Action: DEC_TTL

Decrease TTL value.

If there is no valid RTE_FLOW_ITEM_TYPE_IPV4 or RTE_FLOW_ITEM_TYPE_IPV6 in pattern, Some PMDs will reject rule because behavior will be undefined.

Table 5.79: DEC_TTL

Field
no properties

Action: SET_TTL

Assigns a new TTL value.

If there is no valid RTE_FLOW_ITEM_TYPE_IPV4 or RTE_FLOW_ITEM_TYPE_IPV6 in pattern, Some PMDs will reject rule because behavior will be undefined.

Table 5.80: SET_TTL

Field	Value
ttl_value	new TTL value

Action: SET_MAC_SRC

Set source MAC address.

It must be used with a valid RTE_FLOW_ITEM_TYPE_ETH flow pattern item. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.81: SET_MAC_SRC

Field	Value
mac_addr	MAC address

Action: SET_MAC_DST

Set destination MAC address.

It must be used with a valid RTE_FLOW_ITEM_TYPE_ETH flow pattern item. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.82: SET_MAC_DST

Field	Value
mac_addr	MAC address

Action: INC_TCP_SEQ

Increase sequence number in the outermost TCP header. Value to increase TCP sequence number by is a big-endian 32 bit integer.

Using this action on non-matching traffic will result in undefined behavior.

Action: DEC_TCP_SEQ

Decrease sequence number in the outermost TCP header. Value to decrease TCP sequence number by is a big-endian 32 bit integer.

Using this action on non-matching traffic will result in undefined behavior.

Action: INC_TCP_ACK

Increase acknowledgment number in the outermost TCP header. Value to increase TCP acknowledgment number by is a big-endian 32 bit integer.

Using this action on non-matching traffic will result in undefined behavior.

Action: DEC_TCP_ACK

Decrease acknowledgment number in the outermost TCP header. Value to decrease TCP acknowledgment number by is a big-endian 32 bit integer.

Using this action on non-matching traffic will result in undefined behavior.

Action: SET_TAG

Set Tag.

Tag is a transient data used during flow matching. This is not delivered to application. Multiple tags are supported by specifying index.

Table 5.83: SET_TAG

Field	Value
data	32 bit tag value
mask	bit-mask applies to "data"
index	index of tag to set

Action: SET_META

Set metadata. Item `META` matches metadata.

Metadata set by mbuf metadata field with PKT_TX_DYNF_METADATA flag on egress will be overridden by this action. On ingress, the metadata will be carried by `metadata` dynamic field of `rte_mbuf` which can be accessed by `RTE_FLOW_DYNF_METADATA()`. PKT_RX_DYNF_METADATA flag will be set along with the data.

The mbuf dynamic field must be registered by calling `rte_flow_dynf_metadata_register()` prior to use `SET_META` action.

Altering partial bits is supported with `mask`. For bits which have never been set, unpredictable value will be seen depending on driver implementation. For loopback/hairpin packet, metadata set on Rx/Tx may or may not be propagated to the other path depending on HW capability.

Table 5.84: SET_META

Field	Value
data	32 bit metadata value
mask	bit-mask applies to "data"

Action: SET_IPV4_DSCP

Set IPv4 DSCP.

Modify DSCP in IPv4 header.

It must be used with RTE_FLOW_ITEM_TYPE_IPV4 in pattern. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.85: SET_IPV4_DSCP

Field	Value
dscp	DSCP in low 6 bits, rest ignore

Action: SET_IPV6_DSCP

Set IPv6 DSCP.

Modify DSCP in IPv6 header.

It must be used with RTE_FLOW_ITEM_TYPE_IPV6 in pattern. Otherwise, RTE_FLOW_ERROR_TYPE_ACTION error will be returned.

Table 5.86: SET_IPV6_DSCP

Field	Value
dscp	DSCP in low 6 bits, rest ignore

Action: AGE

Set ageing timeout configuration to a flow.

Event RTE_ETH_EVENT_FLOW_AGED will be reported if timeout passed without any matching on the flow.

Table 5.87: AGE

Field	Value
timeout	24 bits timeout value
reserved	8 bits reserved, must be zero
context	user input flow context

Negative types

All specified pattern items (`enum rte_flow_item_type`) and actions (`enum rte_flow_action_type`) use positive identifiers.

The negative space is reserved for dynamic types generated by PMDs during run-time. PMDs may encounter them as a result but must not accept negative identifiers they are not aware of.

A method to generate them remains to be defined.

Planned types

Pattern item types will be added as new protocols are implemented.

Variable headers support through dedicated pattern items, for example in order to match specific IPv4 options and IPv6 extension headers would be stacked after IPv4/IPv6 items.

Other action types are planned but are not defined yet. These include the ability to alter packet data in several ways, such as performing encapsulation/decapsulation of tunnel headers.

5.12.3 Rules management

A rather simple API with few functions is provided to fully manage flow rules.

Each created flow rule is associated with an opaque, PMD-specific handle pointer. The application is responsible for keeping it until the rule is destroyed.

Flows rules are represented by `struct rte_flow` objects.

Validation

Given that expressing a definite set of device capabilities is not practical, a dedicated function is provided to check if a flow rule is supported and can be created.

```
int
rte_flow_validate(uint16_t port_id,
                  const struct rte_flow_attr *attr,
                  const struct rte_flow_item pattern[],
                  const struct rte_flow_action actions[],
                  struct rte_flow_error *error);
```

The flow rule is validated for correctness and whether it could be accepted by the device given sufficient resources. The rule is checked against the current device mode and queue configuration. The flow rule may also optionally be validated against existing flow rules and device resources. This function has no effect on the target device.

The returned value is guaranteed to remain valid only as long as no successful calls to `rte_flow_create()` or `rte_flow_destroy()` are made in the meantime and no device parameter affecting flow rules in any way are modified, due to possible collisions or resource limitations (although in such cases `EINVAL` should not be returned).

Arguments:

- `port_id`: port identifier of Ethernet device.
- `attr`: flow rule attributes.

- `pattern`: pattern specification (list terminated by the END pattern item).
- `actions`: associated actions (list terminated by the END action).
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

- 0 if flow rule is valid and can be created. A negative errno value otherwise (`rte_errno` is also set), the following errors are defined.
- `-ENOSYS`: underlying device does not support this functionality.
- `-EINVAL`: unknown or invalid rule specification.
- `-ENOTSUP`: valid but unsupported rule specification (e.g. partial bit-masks are unsupported).
- `EEXIST`: collision with an existing rule. Only returned if device supports flow rule collision checking and there was a flow rule collision. Not receiving this return code is no guarantee that creating the rule will not fail due to a collision.
- `ENOMEM`: not enough memory to execute the function, or if the device supports resource validation, resource limitation on the device.
- `-EBUSY`: action cannot be performed due to busy device resources, may succeed if the affected queues or even the entire port are in a stopped state (see `rte_eth_dev_rx_queue_stop()` and `rte_eth_dev_stop()`).

Creation

Creating a flow rule is similar to validating one, except the rule is actually created and a handle returned.

```
struct rte_flow *
rte_flow_create(uint16_t port_id,
                const struct rte_flow_attr *attr,
                const struct rte_flow_item pattern[],
                const struct rte_flow_action *actions[],
                struct rte_flow_error *error);
```

Arguments:

- `port_id`: port identifier of Ethernet device.
- `attr`: flow rule attributes.
- `pattern`: pattern specification (list terminated by the END pattern item).
- `actions`: associated actions (list terminated by the END action).
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

A valid handle in case of success, NULL otherwise and `rte_errno` is set to the positive version of one of the error codes defined for `rte_flow_validate()`.

Destruction

Flow rules destruction is not automatic, and a queue or a port should not be released if any are still attached to them. Applications must take care of performing this step before releasing resources.

```
int
rte_flow_destroy(uint16_t port_id,
                 struct rte_flow *flow,
                 struct rte_flow_error *error);
```

Failure to destroy a flow rule handle may occur when other flow rules depend on it, and destroying it would result in an inconsistent state.

This function is only guaranteed to succeed if handles are destroyed in reverse order of their creation.

Arguments:

- `port_id`: port identifier of Ethernet device.
- `flow`: flow rule handle to destroy.
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

- 0 on success, a negative errno value otherwise and `rte_errno` is set.

Flush

Convenience function to destroy all flow rule handles associated with a port. They are released as with successive calls to `rte_flow_destroy()`.

```
int
rte_flow_flush(uint16_t port_id,
               struct rte_flow_error *error);
```

In the unlikely event of failure, handles are still considered destroyed and no longer valid but the port must be assumed to be in an inconsistent state.

Arguments:

- `port_id`: port identifier of Ethernet device.
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

- 0 on success, a negative errno value otherwise and `rte_errno` is set.

Query

Query an existing flow rule.

This function allows retrieving flow-specific data such as counters. Data is gathered by special actions which must be present in the flow rule definition.

```
int
rte_flow_query(uint16_t port_id,
               struct rte_flow *flow,
               const struct rte_flow_action *action,
               void *data,
               struct rte_flow_error *error);
```

Arguments:

- `port_id`: port identifier of Ethernet device.
- `flow`: flow rule handle to query.
- `action`: action to query, this must match prototype from flow rule.
- `data`: pointer to storage for the associated query data type.
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

- 0 on success, a negative errno value otherwise and `rte_errno` is set.

5.12.4 Flow isolated mode

The general expectation for ingress traffic is that flow rules process it first; the remaining unmatched or pass-through traffic usually ends up in a queue (with or without RSS, locally or in some sub-device instance) depending on the global configuration settings of a port.

While fine from a compatibility standpoint, this approach makes drivers more complex as they have to check for possible side effects outside of this API when creating or destroying flow rules. It results in a more limited set of available rule types due to the way device resources are assigned (e.g. no support for the RSS action even on capable hardware).

Given that nonspecific traffic can be handled by flow rules as well, isolated mode is a means for applications to tell a driver that ingress on the underlying port must be injected from the defined flow rules only; that no default traffic is expected outside those rules.

This has the following benefits:

- Applications get finer-grained control over the kind of traffic they want to receive (no traffic by default).
- More importantly they control at what point nonspecific traffic is handled relative to other flow rules, by adjusting priority levels.
- Drivers can assign more hardware resources to flow rules and expand the set of supported rule types.

Because toggling isolated mode may cause profound changes to the ingress processing path of a driver, it may not be possible to leave it once entered. Likewise, existing flow rules or global configuration settings may prevent a driver from entering isolated mode.

Applications relying on this mode are therefore encouraged to toggle it as soon as possible after device initialization, ideally before the first call to `rte_eth_dev_configure()` to avoid possible failures due to conflicting settings.

Once effective, the following functionality has no effect on the underlying port and may return errors such as `ENOTSUP` ("not supported"):

- Toggling promiscuous mode.
- Toggling allmulticast mode.
- Configuring MAC addresses.
- Configuring multicast addresses.
- Configuring VLAN filters.
- Configuring Rx filters through the legacy API (e.g. FDIR).
- Configuring global RSS settings.

```
int
rte_flow_isolate(uint16_t port_id, int set, struct rte_flow_error *error);
```

Arguments:

- `port_id`: port identifier of Ethernet device.
- `set`: nonzero to enter isolated mode, attempt to leave it otherwise.
- `error`: perform verbose error reporting if not NULL. PMDs initialize this structure in case of error only.

Return values:

- 0 on success, a negative errno value otherwise and `rte_errno` is set.

5.12.5 Verbose error reporting

The defined *errno* values may not be accurate enough for users or application developers who want to investigate issues related to flow rules management. A dedicated error object is defined for this purpose:

```
enum rte_flow_error_type {
    RTE_FLOW_ERROR_TYPE_NONE, /**< No error. */
    RTE_FLOW_ERROR_TYPE_UNSPECIFIED, /**< Cause unspecified. */
    RTE_FLOW_ERROR_TYPE_HANDLE, /**< Flow rule (handle). */
    RTE_FLOW_ERROR_TYPE_ATTR_GROUP, /**< Group field. */
    RTE_FLOW_ERROR_TYPE_ATTR_PRIORITY, /**< Priority field. */
    RTE_FLOW_ERROR_TYPE_ATTR_INGRESS, /**< Ingress field. */
    RTE_FLOW_ERROR_TYPE_ATTR_EGRESS, /**< Egress field. */
    RTE_FLOW_ERROR_TYPE_ATTR, /**< Attributes structure. */
    RTE_FLOW_ERROR_TYPE_ITEM_NUM, /**< Pattern length. */
    RTE_FLOW_ERROR_TYPE_ITEM, /**< Specific pattern item. */
    RTE_FLOW_ERROR_TYPE_ACTION_NUM, /**< Number of actions. */
    RTE_FLOW_ERROR_TYPE_ACTION, /**< Specific action. */
};
```

(continues on next page)

(continued from previous page)

```
struct rte_flow_error {
    enum rte_flow_error_type type; /**< Cause field and error types. */
    const void *cause; /**< Object responsible for the error. */
    const char *message; /**< Human-readable error message. */
};
```

Error type `RTE_FLOW_ERROR_TYPE_NONE` stands for no error, in which case remaining fields can be ignored. Other error types describe the type of the object pointed by `cause`.

If non-NULL, `cause` points to the object responsible for the error. For a flow rule, this may be a pattern item or an individual action.

If non-NULL, `message` provides a human-readable error message.

This object is normally allocated by applications and set by PMDs in case of error, the message points to a constant string which does not need to be freed by the application, however its pointer can be considered valid only as long as its associated DPDK port remains configured. Closing the underlying device or unloading the PMD invalidates it.

5.12.6 Helpers

Error initializer

```
static inline int
rte_flow_error_set(struct rte_flow_error *error,
                   int code,
                   enum rte_flow_error_type type,
                   const void *cause,
                   const char *message);
```

This function initializes `error` (if non-NULL) with the provided parameters and sets `rte_errno` to `code`. A negative error `code` is then returned.

Object conversion

```
int
rte_flow_conv(enum rte_flow_conv_op op,
              void *dst,
              size_t size,
              const void *src,
              struct rte_flow_error *error);
```

Convert `src` to `dst` according to operation `op`. Possible operations include:

- Attributes, pattern item or action duplication.
- Duplication of an entire pattern or list of actions.
- Duplication of a complete flow rule description.
- Pattern item or action name retrieval.

5.12.7 Caveats

- DPDK does not keep track of flow rules definitions or flow rule objects automatically. Applications may keep track of the former and must keep track of the latter. PMDs may also do it for internal needs, however this must not be relied on by applications.
- Flow rules are not maintained between successive port initializations. An application exiting without releasing them and restarting must re-create them from scratch.
- API operations are synchronous and blocking (`EAGAIN` cannot be returned).
- There is no provision for re-entrancy/multi-thread safety, although nothing should prevent different devices from being configured at the same time. PMDs may protect their control path functions accordingly.
- Stopping the data path (TX/RX) should not be necessary when managing flow rules. If this cannot be achieved naturally or with workarounds (such as temporarily replacing the burst function pointers), an appropriate error code must be returned (`EBUSY`).
- PMDs, not applications, are responsible for maintaining flow rules configuration when stopping and restarting a port or performing other actions which may affect them. They can only be destroyed explicitly by applications.

For devices exposing multiple ports sharing global settings affected by flow rules:

- All ports under DPDK control must behave consistently, PMDs are responsible for making sure that existing flow rules on a port are not affected by other ports.
- Ports not under DPDK control (unaffected or handled by other applications) are user's responsibility. They may affect existing flow rules and cause undefined behavior. PMDs aware of this may prevent flow rules creation altogether in such cases.

5.12.8 PMD interface

The PMD interface is defined in `rte_flow_driver.h`. It is not subject to API/ABI versioning constraints as it is not exposed to applications and may evolve independently.

It is currently implemented on top of the legacy filtering framework through filter type *RTE_ETH_FILTER_GENERIC* that accepts the single operation *RTE_ETH_FILTER_GET* to return PMD-specific *rte_flow* callbacks wrapped inside `struct rte_flow_ops`.

This overhead is temporarily necessary in order to keep compatibility with the legacy filtering framework, which should eventually disappear.

- PMD callbacks implement exactly the interface described in *Rules management*, except for the port ID argument which has already been converted to a pointer to the underlying `struct rte_eth_dev`.
- Public API functions do not process flow rules definitions at all before calling PMD functions (no basic error checking, no validation whatsoever). They only make sure these callbacks are non-NULL or return the `ENOSYS` (function not supported) error.

This interface additionally defines the following helper function:

- `rte_flow_ops_get()`: get generic flow operations structure from a port.

More will be added over time.

5.12.9 Device compatibility

No known implementation supports all the described features.

Unsupported features or combinations are not expected to be fully emulated in software by PMDs for performance reasons. Partially supported features may be completed in software as long as hardware performs most of the work (such as queue redirection and packet recognition).

However PMDs are expected to do their best to satisfy application requests by working around hardware limitations as long as doing so does not affect the behavior of existing flow rules.

The following sections provide a few examples of such cases and describe how PMDs should handle them, they are based on limitations built into the previous APIs.

Global bit-masks

Each flow rule comes with its own, per-layer bit-masks, while hardware may support only a single, device-wide bit-mask for a given layer type, so that two IPv4 rules cannot use different bit-masks.

The expected behavior in this case is that PMDs automatically configure global bit-masks according to the needs of the first flow rule created.

Subsequent rules are allowed only if their bit-masks match those, the `EEXIST` error code should be returned otherwise.

Unsupported layer types

Many protocols can be simulated by crafting patterns with the *Item: RAW* type.

PMDs can rely on this capability to simulate support for protocols with headers not directly recognized by hardware.

`ANY` pattern item

This pattern item stands for anything, which can be difficult to translate to something hardware would understand, particularly if followed by more specific types.

Consider the following pattern:

Table 5.88: Pattern with ANY as L3

Index	Item		
0	ETHER		
1	ANY	`num`	1
2	TCP		
3	END		

Knowing that TCP does not make sense with something other than IPv4 and IPv6 as L3, such a pattern may be translated to two flow rules instead:

Table 5.89: ANY replaced with IPV4

Index	Item
0	ETHER
1	IPV4 (zeroed mask)
2	TCP
3	END

Table 5.90: ANY replaced with IPV6

Index	Item
0	ETHER
1	IPV6 (zeroed mask)
2	TCP
3	END

Note that as soon as a ANY rule covers several layers, this approach may yield a large number of hidden flow rules. It is thus suggested to only support the most common scenarios (anything as L2 and/or L3).

Unsupported actions

- When combined with *Action: QUEUE*, packet counting (*Action: COUNT*) and tagging (*Action: MARK* or *Action: FLAG*) may be implemented in software as long as the target queue is used by a single rule.
- When a single target queue is provided, *Action: RSS* can also be implemented through *Action: QUEUE*.

Flow rules priority

While it would naturally make sense, flow rules cannot be assumed to be processed by hardware in the same order as their creation for several reasons:

- They may be managed internally as a tree or a hash table instead of a list.
- Removing a flow rule before adding another one can either put the new rule at the end of the list or reuse a freed entry.
- Duplication may occur when packets are matched by several rules.

For overlapping rules (particularly in order to use *Action: PASSTHRU*) predictable behavior is only guaranteed by using different priority levels.

Priority levels are not necessarily implemented in hardware, or may be severely limited (e.g. a single priority bit).

For these reasons, priority levels may be implemented purely in software by PMDs.

- For devices expecting flow rules to be added in the correct order, PMDs may destroy and re-create existing rules after adding a new one with a higher priority.
- A configurable number of dummy or empty rules can be created at initialization time to save high priority slots for later.
- In order to save priority levels, PMDs may evaluate whether rules are likely to collide and adjust their priority accordingly.

5.12.10 Future evolutions

- A device profile selection function which could be used to force a permanent profile instead of relying on its automatic configuration based on existing flow rules.
- A method to optimize *rte_flow* rules with specific pattern items and action types generated on the fly by PMDs. DPDK should assign negative numbers to these in order to not collide with the existing types. See *Negative types*.
- Adding specific egress pattern items and actions as described in *Attribute: Traffic direction*.
- Optional software fallback when PMDs are unable to handle requested flow rules so applications do not have to implement their own.

5.13 Switch Representation within DPDK Applications

5.13.1 Introduction

Network adapters with multiple physical ports and/or SR-IOV capabilities usually support the offload of traffic steering rules between their virtual functions (VFs), physical functions (PFs) and ports.

Like for standard Ethernet switches, this involves a combination of automatic MAC learning and manual configuration. For most purposes it is managed by the host system and fully transparent to users and applications.

On the other hand, applications typically found on hypervisors that process layer 2 (L2) traffic (such as OVS) need to steer traffic themselves according on their own criteria.

Without a standard software interface to manage traffic steering rules between VFs, PFs and the various physical ports of a given device, applications cannot take advantage of these offloads; software processing is mandatory even for traffic which ends up re-injected into the device it originates from.

This document describes how such steering rules can be configured through the DPDK flow API (**rte_flow**), with emphasis on the SR-IOV use case (PF/VF steering) using a single physical port for clarity, however the same logic applies to any number of ports without necessarily involving SR-IOV.

5.13.2 Port Representors

In many cases, traffic steering rules cannot be determined in advance; applications usually have to process a bit of traffic in software before thinking about offloading specific flows to hardware.

Applications therefore need the ability to receive and inject traffic to various device endpoints (other VFs, PFs or physical ports) before connecting them together. Device drivers must provide means to hook the "other end" of these endpoints and to refer them when configuring flow rules.

This role is left to so-called "port representors" (also known as "VF representors" in the specific context of VFs), which are to DPDK what the Ethernet switch device driver model (**switchdev**)[1] is to Linux, and which can be thought as a software "patch panel" front-end for applications.

[1] Ethernet switch device driver model (switchdev)

- DPDK port representors are implemented as additional virtual Ethernet device (**ethdev**) instances, spawned on an as needed basis through configuration parameters passed to the driver of the underlying device using devargs.

```
-w pci:dbdf,representor=0
-w pci:dbdf,representor=[0-3]
-w pci:dbdf,representor=[0,5-11]
```

- As virtual devices, they may be more limited than their physical counterparts, for instance by exposing only a subset of device configuration callbacks and/or by not necessarily having Rx/Tx capability.
- Among other things, they can be used to assign MAC addresses to the resource they represent.
- Applications can tell port representors apart from other physical of virtual port by checking the dev_flags field within their device information structure for the RTE_ETH_DEV_REPRESENTOR bit-field.

```
struct rte_eth_dev_info {
    ...
    uint32_t dev_flags; /**< Device flags */
    ...
};
```

- The device or group relationship of ports can be discovered using the switch `domain_id` field within the devices switch information structure. By default the switch `domain_id` of a port will be `RTE_ETH_DEV_SWITCH_DOMAIN_ID_INVALID` to indicate that the port doesn't support the concept of a switch domain, but ports which do support the concept will be allocated a unique switch `domain_id`, ports within the same switch domain will share the same `domain_id`. The switch `port_id` is used to specify the port_id in terms of the switch, so in the case of SR-IOV devices the switch `port_id` would represent the virtual function identifier of the port.

```
/**
 * Ethernet device associated switch information
 */
struct rte_eth_switch_info {
    const char *name; /**< switch name */
    uint16_t domain_id; /**< switch domain id */
    uint16_t port_id; /**< switch port id */
};
```

5.13.3 Basic SR-IOV

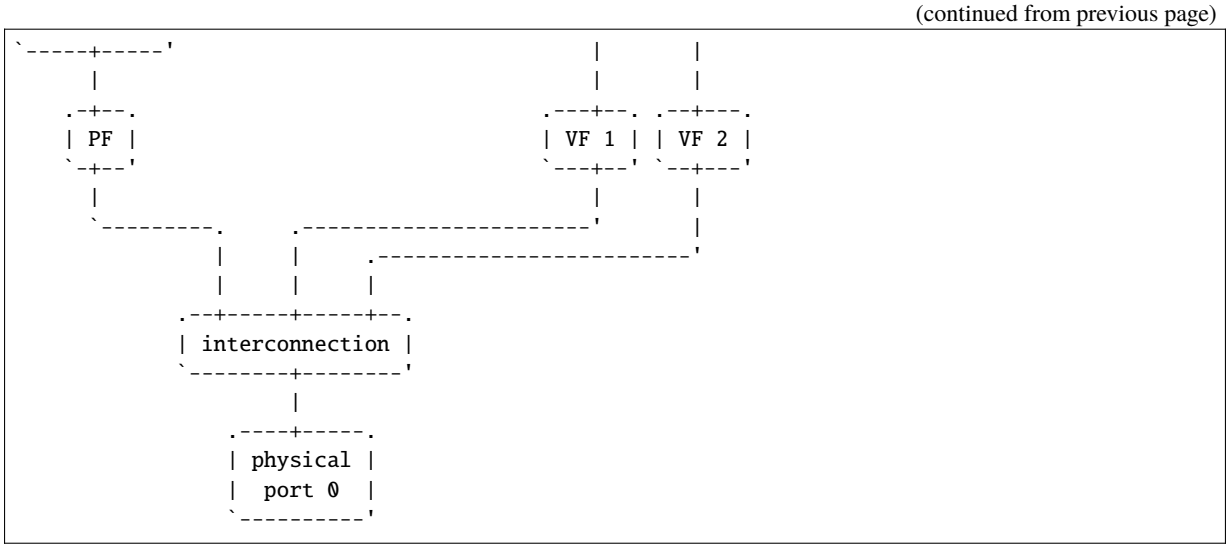
"Basic" in the sense that it is not managed by applications, which nonetheless expect traffic to flow between the various endpoints and the outside as if everything was linked by an Ethernet hub.

The following diagram pictures a setup involving a device with one PF, two VFs and one shared physical port

```
   .-------------.                .-------------. .-------------.
   | hypervisor  |                |    VM 1     | |    VM 2     |
   | application |                | application | | application |
   `--+----------'                `----------+--' `--+----------'
      |                                      |       |
.-----+-----.                                |       |
| port_id 3 |                                |       |
```

(continues on next page)

(continued from previous page)



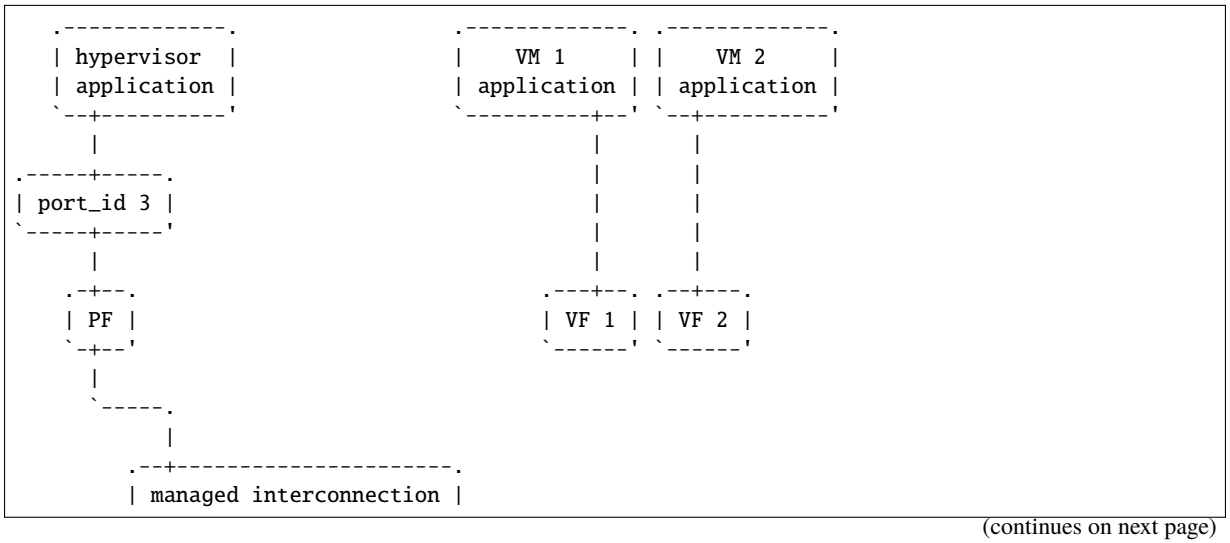
- A DPDK application running on the hypervisor owns the PF device, which is arbitrarily assigned port index 3.
- Both VFs are assigned to VMs and used by unknown applications; they may be DPDK-based or anything else.
- Interconnection is not necessarily done through a true Ethernet switch and may not even exist as a separate entity. The role of this block is to show that something brings PF, VFs and physical ports together and enables communication between them, with a number of built-in restrictions.

Subsequent sections in this document describe means for DPDK applications running on the hypervisor to freely assign specific flows between PF, VFs and physical ports based on traffic properties, by managing this interconnection.

5.13.4 Controlled SR-IOV

Initialization

When a DPDK application gets assigned a PF device and is deliberately not started in *basic SR-IOV* mode, any traffic coming from physical ports is received by PF according to default rules, while VFs remain isolated.



(continues on next page)

(continued from previous page)

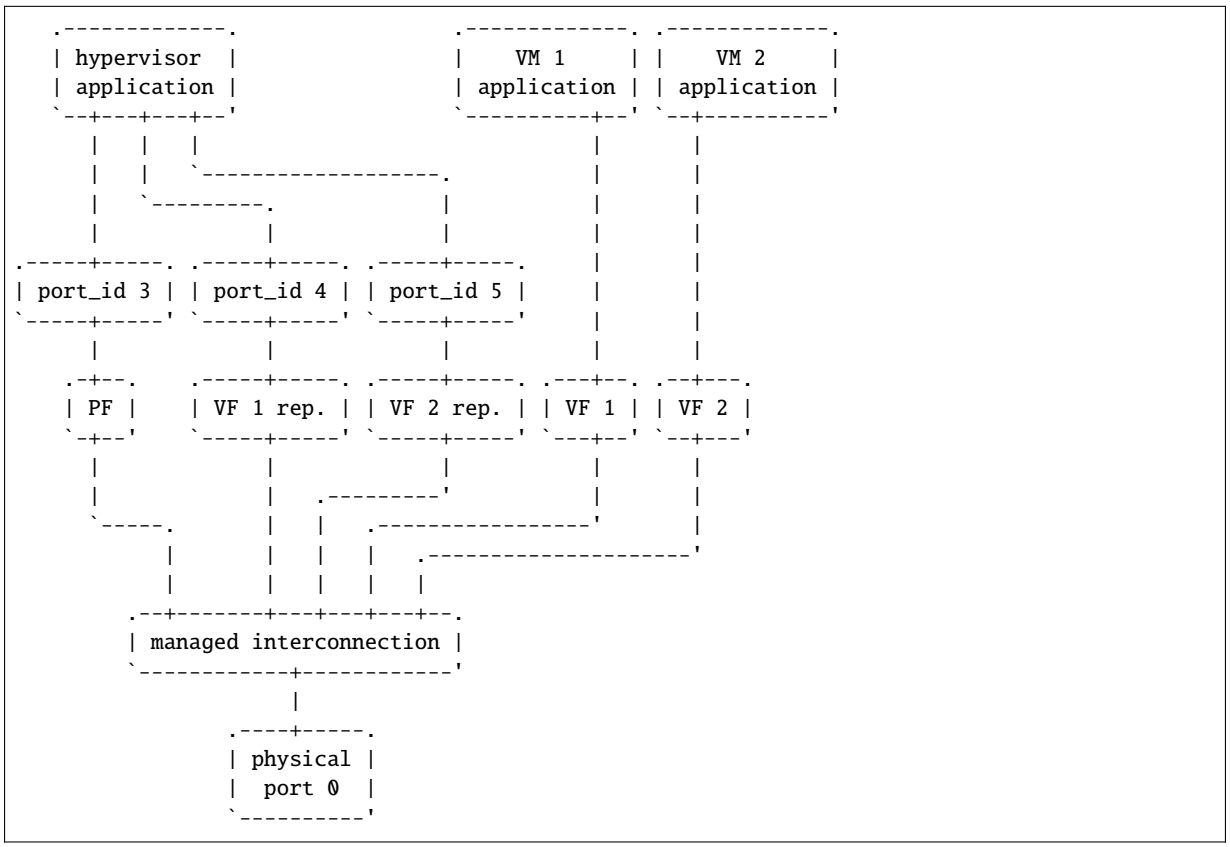
In this mode, interconnection must be configured by the application to enable VF communication, for instance by explicitly directing traffic with a given destination MAC address to VF 1 and allowing that with the same source MAC address to come out of it.

For this to work, hypervisor applications need a way to refer to either VF 1 or VF 2 in addition to the PF. This is addressed by *VF representors*.

VF Representors

VF representors are virtual but standard DPDK network devices (albeit with limited capabilities) created by PMDs when managing a PF device.

Since they represent VF instances used by other applications, configuring them (e.g. assigning a MAC address or setting up promiscuous mode) affects interconnection accordingly. If supported, they may also be used as two-way communication ports with VFs (assuming **switchdev** topology)



- VF representors are assigned arbitrary port indices 4 and 5 in the hypervisor application and are respectively associated with VF 1 and VF 2.
- They can't be dissociated; even if VF 1 and VF 2 were not connected, representors could still be used for configuration.

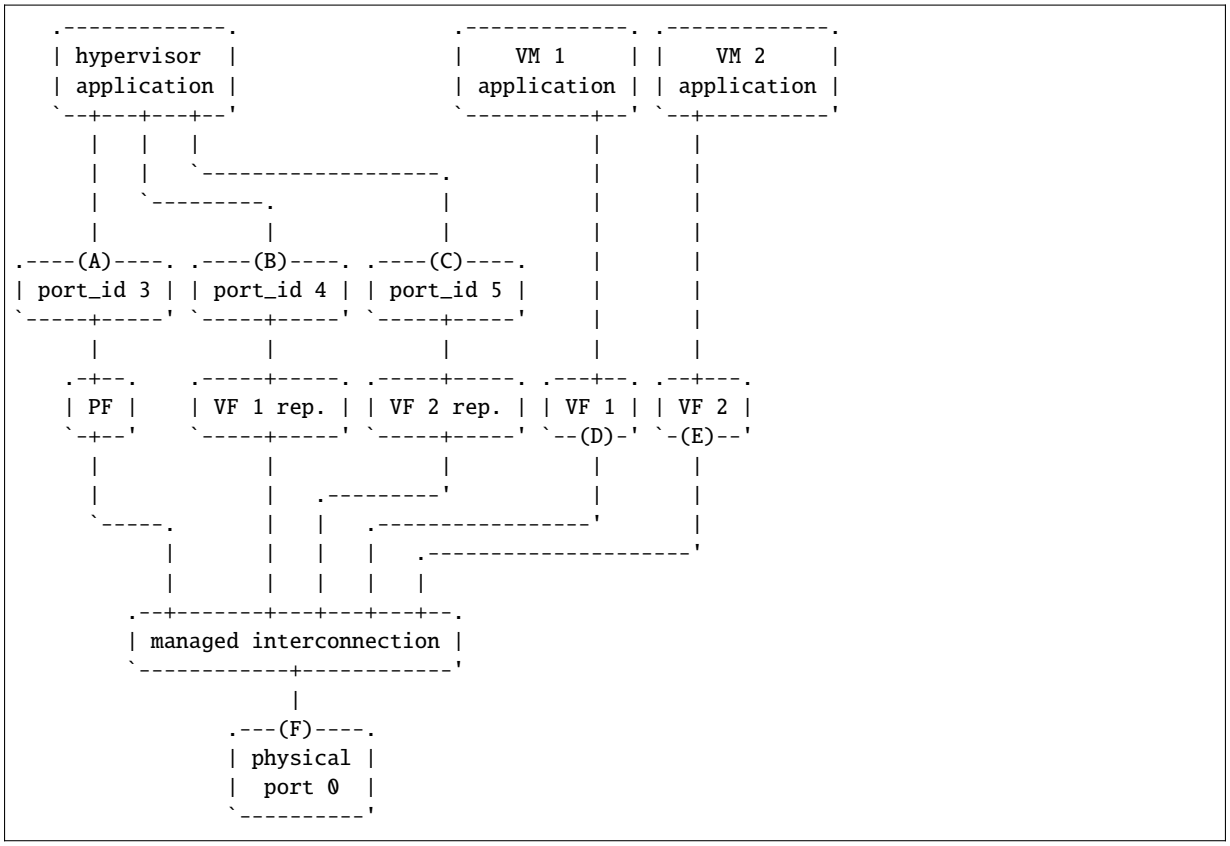
- In this context, port index 3 can be thought as a representor for physical port 0.

As previously described, the "interconnection" block represents a logical concept. Interconnection occurs when hardware configuration enables traffic flows from one place to another (e.g. physical port 0 to VF 1) according to some criteria.

This is discussed in more detail in *traffic steering*.

Traffic Steering

In the following diagram, each meaningful traffic origin or endpoint as seen by the hypervisor application is tagged with a unique letter from A to F.



- **A**: PF device.
- **B**: port representor for VF 1.
- **C**: port representor for VF 2.
- **D**: VF 1 proper.
- **E**: VF 2 proper.
- **F**: physical port.

Although uncommon, some devices do not enforce a one to one mapping between PF and physical ports. For instance, by default all ports of **mlx4** adapters are available to all their PF/VF instances, in which case additional ports appear next to **F** in the above diagram.

Assuming no interconnection is provided by default in this mode, setting up a *basic SR-IOV* configuration involving physical port 0 could be broken down as:

PF:

- **A to F**: let everything through.
- **F to A**: PF MAC as destination.

VF 1:

- **A to D**, **E to D** and **F to D**: VF 1 MAC as destination.
- **D to A**: VF 1 MAC as source and PF MAC as destination.
- **D to E**: VF 1 MAC as source and VF 2 MAC as destination.
- **D to F**: VF 1 MAC as source.

VF 2:

- **A to E**, **D to E** and **F to E**: VF 2 MAC as destination.
- **E to A**: VF 2 MAC as source and PF MAC as destination.
- **E to D**: VF 2 MAC as source and VF 1 MAC as destination.
- **E to F**: VF 2 MAC as source.

Devices may additionally support advanced matching criteria such as IPv4/IPv6 addresses or TCP/UDP ports.

The combination of matching criteria with target endpoints fits well with **rte_flow**[6], which expresses flow rules as combinations of patterns and actions.

Enhancing **rte_flow** with the ability to make flow rules match and target these endpoints provides a standard interface to manage their interconnection without introducing new concepts and whole new API to implement them. This is described in *flow API (rte_flow)*.

5.13.5 Flow API (rte_flow)

Extensions

Compared to creating a brand new dedicated interface, **rte_flow** was deemed flexible enough to manage representor traffic only with minor extensions:

- Using physical ports, PF, VF or port representors as targets.
- Affecting traffic that is not necessarily addressed to the DPDK port ID a flow rule is associated with (e.g. forcing VF traffic redirection to PF).

For advanced uses:

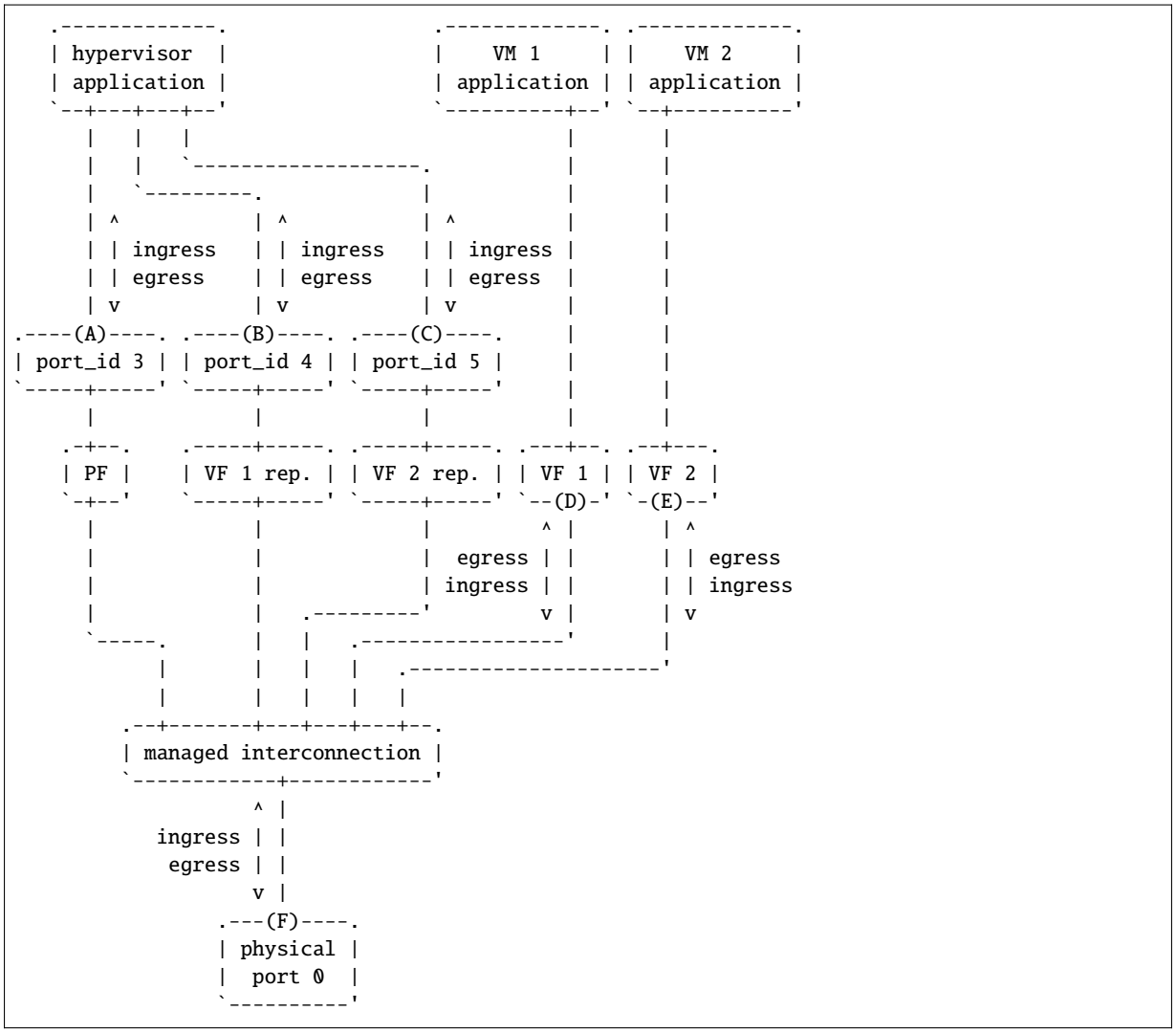
- Rule-based packet counters.
- The ability to combine several identical actions for traffic duplication (e.g. VF representor in addition to a physical port).
- Dedicated actions for traffic encapsulation / decapsulation before reaching an endpoint.

[6] *Generic flow API (rte_flow)*

Traffic Direction

From an application standpoint, "ingress" and "egress" flow rule attributes apply to the DPDK port ID they are associated with. They select a traffic direction for matching patterns, but have no impact on actions.

When matching traffic coming from or going to a different place than the immediate port ID a flow rule is associated with, these attributes keep their meaning while applying to the chosen origin, as highlighted by the following diagram



Ingress and egress are defined as relative to the application creating the flow rule.

For instance, matching traffic sent by VM 2 would be done through an ingress flow rule on VF 2 (**E**). Likewise for incoming traffic on physical port (**F**). This also applies to **C** and **A** respectively.

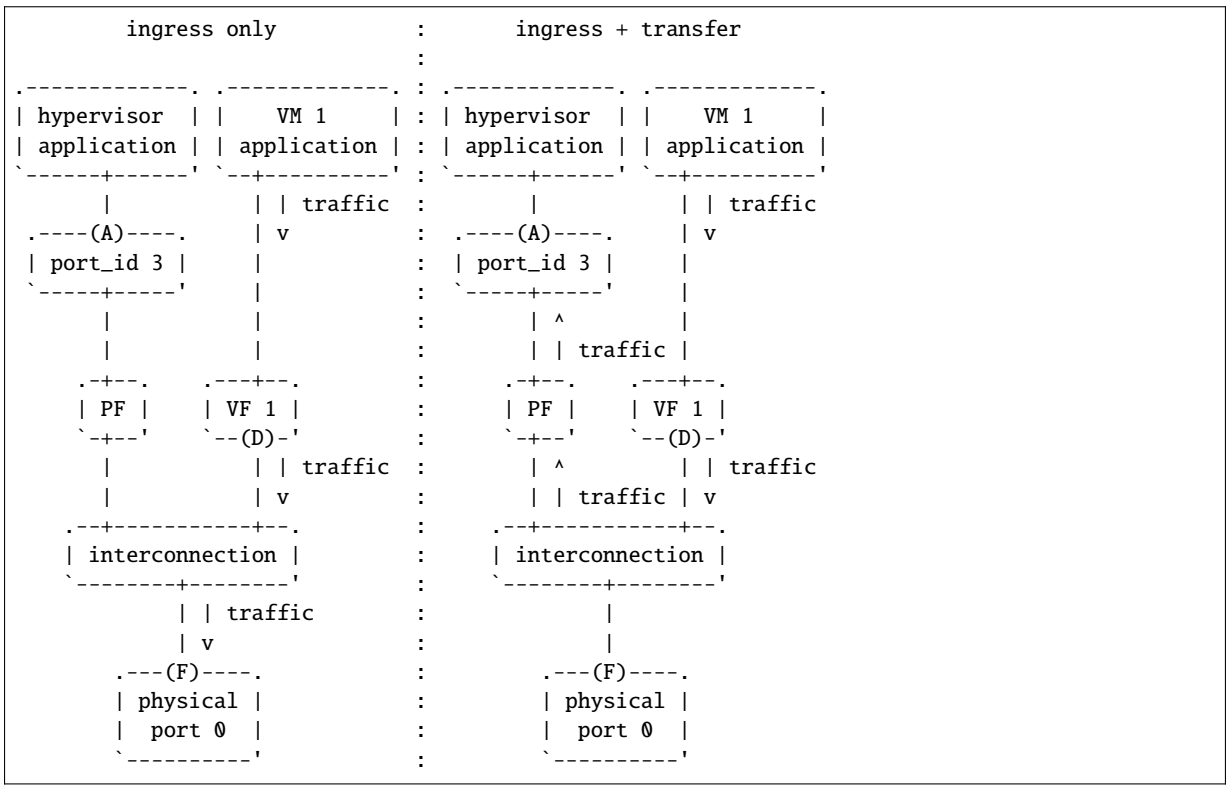
Transferring Traffic

Without Port Representors

Traffic direction describes how an application could match traffic coming from or going to a specific place reachable from a DPDK port ID. This makes sense when the traffic in question is normally seen (i.e. sent or received) by the application creating the flow rule (e.g. as in "redirect all traffic coming from VF 1 to local queue 6").

However this does not force such traffic to take a specific route. Creating a flow rule on **A** matching traffic coming from **D** is only meaningful if it can be received by **A** in the first place, otherwise doing so simply has no effect.

A new flow rule attribute named "transfer" is necessary for that. Combining it with "ingress" or "egress" and a specific origin requests a flow rule to be applied at the lowest level



With "ingress" only, traffic is matched on **A** thus still goes to physical port **F** by default

```
testpmd> flow create 3 ingress pattern vf id is 1 / end
          actions queue index 6 / end
```

With "ingress + transfer", traffic is matched on **D** and is therefore successfully assigned to queue 6 on **A**

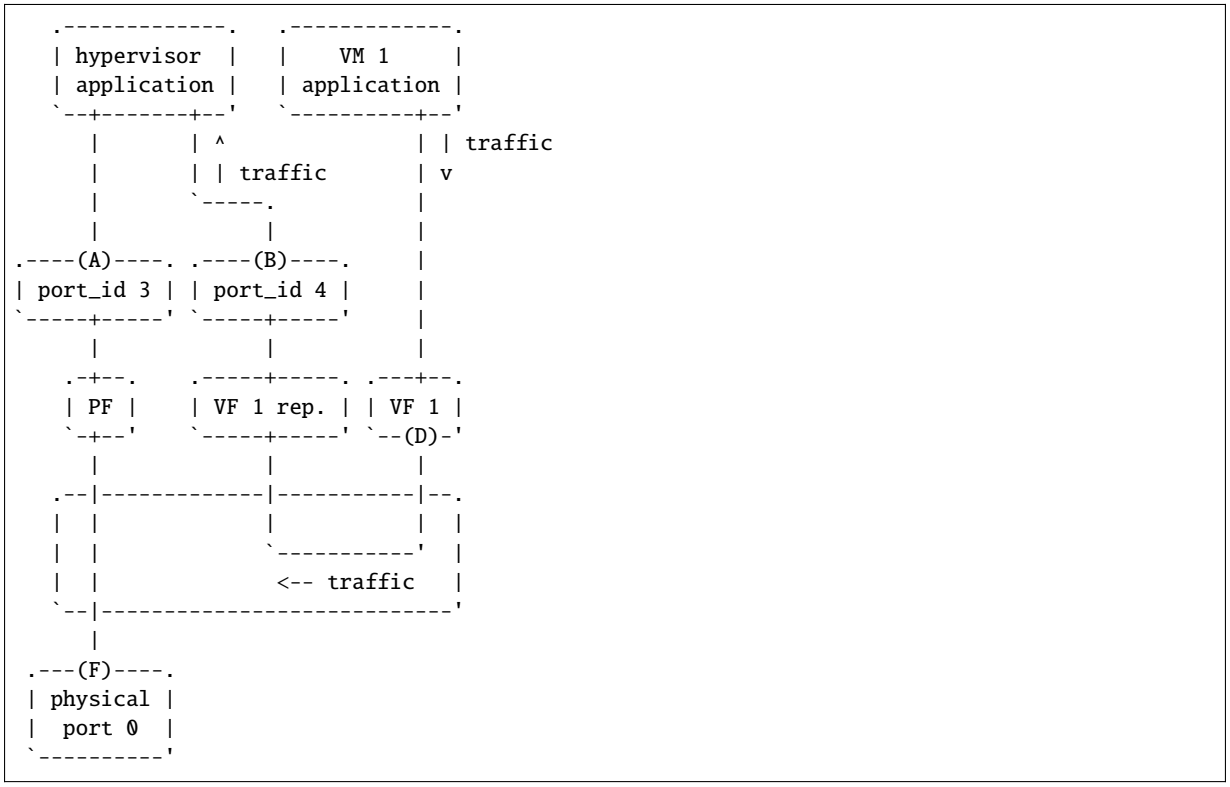
```
testpmd> flow create 3 ingress transfer pattern vf id is 1 / end
         actions queue index 6 / end
```

With Port Representors

When port representors exist, implicit flow rules with the "transfer" attribute (described in *without port representors*) are be assumed to exist between them and their represented resources. These may be immutable.

In this case, traffic is received by default through the representor and neither the "transfer" attribute nor traffic origin in flow rule patterns are necessary. They simply have to be created on the representor port directly and may target a different representor as described in *PORT_ID action*.

Implicit traffic flow with port representor



Pattern Items And Actions

PORT Pattern Item

Matches traffic originating from (ingress) or going to (egress) a physical port of the underlying device.

Using this pattern item without specifying a port index matches the physical port associated with the current DPDK port ID by default. As described in *traffic steering*, specifying it should be rarely needed.

- Matches **F** in *traffic steering*.

PORT Action

Directs matching traffic to a given physical port index.

- Targets **F** in *traffic steering*.

PORT_ID Pattern Item

Matches traffic originating from (ingress) or going to (egress) a given DPDK port ID.

Normally only supported if the port ID in question is known by the underlying PMD and related to the device the flow rule is created against.

This must not be confused with the *PORT pattern item* which refers to the physical port of a device. `PORT_ID` refers to a `struct rte_eth_dev` object on the application side (also known as "port representor" depending on the kind of underlying device).

- Matches **A**, **B** or **C** in *traffic steering*.

PORT_ID Action

Directs matching traffic to a given DPDK port ID.

Same restrictions as *PORT_ID pattern item*.

- Targets **A**, **B** or **C** in *traffic steering*.

PF Pattern Item

Matches traffic originating from (ingress) or going to (egress) the physical function of the current device.

If supported, should work even if the physical function is not managed by the application and thus not associated with a DPDK port ID. Its behavior is otherwise similar to *PORT_ID pattern item* using PF port ID.

- Matches **A** in *traffic steering*.

PF Action

Directs matching traffic to the physical function of the current device.

Same restrictions as *PF pattern item*.

- Targets **A** in *traffic steering*.

VF Pattern Item

Matches traffic originating from (ingress) or going to (egress) a given virtual function of the current device.

If supported, should work even if the virtual function is not managed by the application and thus not associated with a DPDK port ID. Its behavior is otherwise similar to *PORT_ID pattern item* using VF port ID.

Note this pattern item does not match VF representors traffic which, as separate entities, should be addressed through their own port IDs.

- Matches **D** or **E** in *traffic steering*.

VF Action

Directs matching traffic to a given virtual function of the current device.

Same restrictions as *VF pattern item*.

- Targets **D** or **E** in *traffic steering*.

*_ENCAP actions

These actions are named according to the protocol they encapsulate traffic with (e.g. `VXLAN_ENCAP`) and using specific parameters (e.g. VNI for VXLAN).

While they modify traffic and can be used multiple times (order matters), unlike *PORT_ID action* and friends, they have no impact on steering.

As described in *actions order and repetition* this means they are useless if used alone in an action list, the resulting traffic gets dropped unless combined with either `PASSTHRU` or other endpoint-targeting actions.

*_DECAP actions

They perform the reverse of **_ENCAP actions* by popping protocol headers from traffic instead of pushing them. They can be used multiple times as well.

Note that using these actions on non-matching traffic results in undefined behavior. It is recommended to match the protocol headers to decapsulate on the pattern side of a flow rule in order to use these actions or otherwise make sure only matching traffic goes through.

Actions Order and Repetition

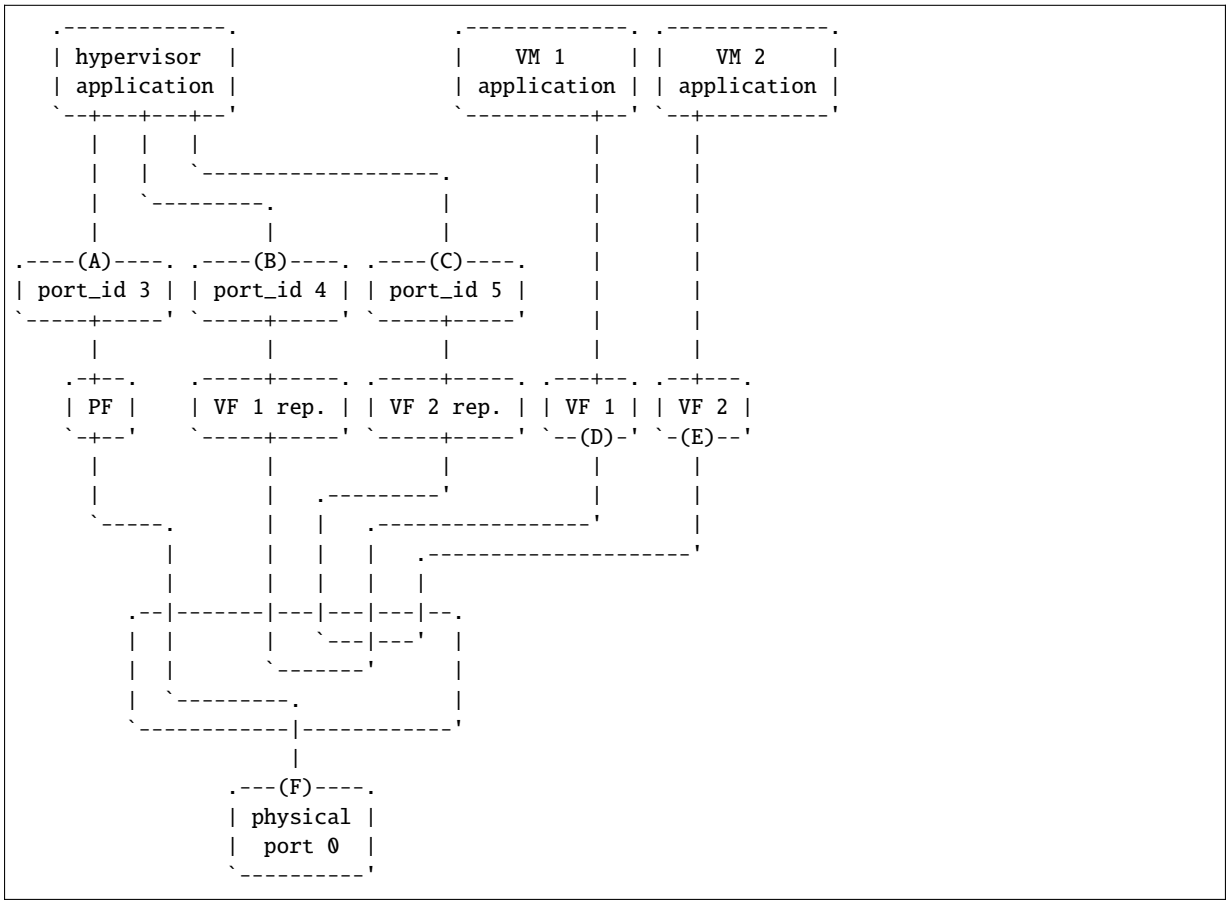
Flow rules are currently restricted to at most a single action of each supported type, performed in an unpredictable order (or all at once). To repeat actions in a predictable fashion, applications have to make rules pass-through and use priority levels.

It's now clear that PMD support for chaining multiple non-terminating flow rules of varying priority levels is prohibitively difficult to implement compared to simply allowing multiple identical actions performed in a defined order by a single flow rule.

- This change is required to support protocol encapsulation offloads and the ability to perform them multiple times (e.g. VLAN then VXLAN).
- It makes the `DUP` action redundant since multiple `QUEUE` actions can be combined for duplication.
- The (non-)terminating property of actions must be discarded. Instead, flow rules themselves must be considered terminating by default (i.e. dropping traffic if there is no specific target) unless a `PASSTHRU` action is also specified.

5.13.6 Switching Examples

This section provides practical examples based on the established testpmd flow command syntax[2], in the context described in *traffic steering*



By default, PF (**A**) can communicate with the physical port it is associated with (**F**), while VF 1 (**D**) and VF 2 (**E**) are isolated and restricted to communicate with the hypervisor application through their respective representors (**B** and **C**) if supported.

Examples in subsequent sections apply to hypervisor applications only and are based on port representors **A**, **B** and **C**.

[2] *Flow syntax*

Associating VF 1 with Physical Port 0

Assign all port traffic (**F**) to VF 1 (**D**) indiscriminately through their representors

```
flow create 3 ingress pattern / end actions port_id id 4 / end
flow create 4 ingress pattern / end actions port_id id 3 / end
```

More practical example with MAC address restrictions

```
flow create 3 ingress
    pattern eth dst is {VF 1 MAC} / end
    actions port_id id 4 / end
```

```
flow create 4 ingress
    pattern eth src is {VF 1 MAC} / end
    actions port_id id 3 / end
```

Sharing Broadcasts

From outside to PF and VFs

```
flow create 3 ingress
   pattern eth dst is ff:ff:ff:ff:ff:ff / end
   actions port_id id 3 / port_id id 4 / port_id id 5 / end
```

Note `port_id id 3` is necessary otherwise only VFs would receive matching traffic.

From PF to outside and VFs

```
flow create 3 egress
   pattern eth dst is ff:ff:ff:ff:ff:ff / end
   actions port / port_id id 4 / port_id id 5 / end
```

From VFs to outside and PF

```
flow create 4 ingress
   pattern eth dst is ff:ff:ff:ff:ff:ff src is {VF 1 MAC} / end
   actions port_id id 3 / port_id id 5 / end

flow create 5 ingress
   pattern eth dst is ff:ff:ff:ff:ff:ff src is {VF 2 MAC} / end
   actions port_id id 4 / port_id id 4 / end
```

Similar `33:33:*` rules based on known MAC addresses should be added for IPv6 traffic.

Encapsulating VF 2 Traffic in VXLAN

Assuming pass-through flow rules are supported

```
flow create 5 ingress
   pattern eth / end
   actions vxlan_encap vni 42 / passthru / end
```

```
flow create 5 egress
   pattern vxlan vni is 42 / end
   actions vxlan_decap / passthru / end
```

Here `passthru` is needed since as described in *actions order and repetition*, flow rules are otherwise terminating; if supported, a rule without a target endpoint will drop traffic.

Without pass-through support, ingress encapsulation on the destination endpoint might not be supported and action list must provide one

```
flow create 5 ingress
   pattern eth src is {VF 2 MAC} / end
   actions vxlan_encap vni 42 / port_id id 3 / end

flow create 3 ingress
   pattern vxlan vni is 42 / end
   actions vxlan_decap / port_id id 5 / end
```

5.14 Traffic Metering and Policing API

5.14.1 Overview

This is the generic API for the Quality of Service (QoS) Traffic Metering and Policing (MTR) of Ethernet devices. This API is agnostic of the underlying HW, SW or mixed HW-SW implementation.

The main features are:

- Part of DPDK rte_ethdev API
- Capability query API
- Metering algorithms: RFC 2697 Single Rate Three Color Marker (srTCM), RFC 2698 and RFC 4115 Two Rate Three Color Marker (trTCM)
- Policer actions (per meter output color): recolor, drop
- Statistics (per policer output color)

5.14.2 Configuration steps

The metering and policing stage typically sits on top of flow classification, which is why the MTR objects are enabled through a special "meter" action.

The MTR objects are created and updated in their own name space (`rte_mtr`) within the `librte_ethdev` library. Whether an MTR object is private to a flow or potentially shared by several flows has to be specified at its creation time.

Once successfully created, an MTR object is hooked into the RX processing path of the Ethernet device by linking it to one or several flows through the dedicated "meter" flow action. One or several "meter" actions can be registered for the same flow. An MTR object can only be destroyed if there are no flows using it.

5.14.3 Run-time processing

Traffic metering determines the color for the current packet (green, yellow, red) based on the previous history for this flow as maintained by the MTR object. The policer can do nothing, override the color the packet or drop the packet. Statistics counters are maintained for MTR object, as configured.

The processing done for each input packet hitting an MTR object is:

- Traffic metering: The packet is assigned a color (the meter output color) based on the previous traffic history reflected in the current state of the MTR object, according to the specific traffic metering algorithm. The traffic metering algorithm can typically work in color aware mode, in which case the input packet already has an initial color (the input color), or in color blind mode, which is equivalent to considering all input packets initially colored as green.
- Policing: There is a separate policer action configured for each meter output color, which can:
  - Drop the packet.
  - Keep the same packet color: the policer output color matches the meter output color (essentially a no-op action).
  - Recolor the packet: the policer output color is set to a different color than the meter output color. The policer output color is the output color of the packet, which is set in the packet meta-data (i.e. struct `rte_mbuf::sched::color`).
- Statistics: The set of counters maintained for each MTR object is configurable and subject to the implementation support. This set includes the number of packets and bytes dropped or passed for each output color.

5.15 Traffic Management API

5.15.1 Overview

This is the generic API for the Quality of Service (QoS) Traffic Management of Ethernet devices, which includes the following main features: hierarchical scheduling, traffic shaping, congestion management, packet marking. This API is agnostic of the underlying HW, SW or mixed HW-SW implementation.

Main features:

- Part of DPDK rte_ethdev API
- Capability query API per port, per hierarchy level and per hierarchy node
- Scheduling algorithms: Strict Priority (SP), Weighed Fair Queuing (WFQ)
- Traffic shaping: single/dual rate, private (per node) and shared (by multiple nodes) shapers
- Congestion management for hierarchy leaf nodes: algorithms of tail drop, head drop, WRED, private (per node) and shared (by multiple nodes) WRED contexts
- Packet marking: IEEE 802.1q (VLAN DEI), IETF RFC 3168 (IPv4/IPv6 ECN for TCP and SCTP), IETF RFC 2597 (IPv4 / IPv6 DSCP)

5.15.2 Capability API

The aim of these APIs is to advertise the capability information (i.e critical parameter values) that the TM implementation (HW/SW) is able to support for the application. The APIs supports the information disclosure at the TM level, at any hierarchical level of the TM and at any node level of the specific hierarchical level. Such information helps towards rapid understanding of whether a specific implementation does meet the needs to the user application.

At the TM level, users can get high level idea with the help of various parameters such as maximum number of nodes, maximum number of hierarchical levels, maximum number of shapers, maximum number of private shapers, type of scheduling algorithm (Strict Priority, Weighted Fair Queuing , etc.), etc., supported by the implementation.

Likewise, users can query the capability of the TM at the hierarchical level to have more granular knowledge about the specific level. The various parameters such as maximum number of nodes at the level, maximum number of leaf/non-leaf nodes at the level, type of the shaper(dual rate, single rate) supported at the level if node is non-leaf type etc., are exposed as a result of hierarchical level capability query.

Finally, the node level capability API offers knowledge about the capability supported by the node at any specific level. The information whether the support is available for private shaper, dual rate shaper, maximum and minimum shaper rate, etc. is exposed by node level capability API.

5.15.3 Scheduling Algorithms

The fundamental scheduling algorithms that are supported are Strict Priority (SP) and Weighted Fair Queuing (WFQ). The SP and WFQ algorithms are supported at the level of each node of the scheduling hierarchy, regardless of the node level/position in the tree. The SP algorithm is used to schedule between sibling nodes with different priority, while WFQ is used to schedule between groups of siblings that have the same priority.

Algorithms such as Weighed Round Robin (WRR), byte-level WRR, Deficit WRR (DWRR), etc are considered approximations of the ideal WFQ and are therefore assimilated to WFQ, although an associated implementation-dependent accuracy, performance and resource usage trade-off might exist.

5.15.4 Traffic Shaping

The TM API provides support for single rate and dual rate shapers (rate limiters) for the hierarchy nodes, subject to the specific implementation support being available.

Each hierarchy node has zero or one private shaper (only one node using it) and/or zero, one or several shared shapers (multiple nodes use the same shaper instance). A private shaper is used to perform traffic shaping for a single node, while a shared shaper is used to perform traffic shaping for a group of nodes.

The configuration of private and shared shapers is done through the definition of shaper profiles. Any shaper profile (single rate or dual rate shaper) can be used by one or several shaper instances (either private or shared).

Single rate shapers use a single token bucket. Therefore, single rate shaper is configured by setting the rate of the committed bucket to zero, which effectively disables this bucket. The peak bucket is used to limit the rate and the burst size for the single rate shaper. Dual rate shapers use both the committed and the peak token buckets. The rate of the peak bucket has to be bigger than zero, as well as greater than or equal to the rate of the committed bucket.

5.15.5 Congestion Management

Congestion management is used to control the admission of packets into a packet queue or group of packet queues on congestion. The congestion management algorithms that are supported are: Tail Drop, Head Drop and Weighted Random Early Detection (WRED). They are made available for every leaf node in the hierarchy, subject to the specific implementation supporting them. On request of writing a new packet into the current queue while the queue is full, the Tail Drop algorithm drops the new packet while leaving the queue unmodified, as opposed to the Head Drop* algorithm, which drops the packet at the head of the queue (the oldest packet waiting in the queue) and admits the new packet at the tail of the queue.

The Random Early Detection (RED) algorithm works by proactively dropping more and more input packets as the queue occupancy builds up. When the queue is full or almost full, RED effectively works as Tail Drop. The Weighted RED (WRED) algorithm uses a separate set of RED thresholds for each packet color and uses separate set of RED thresholds for each packet color.

Each hierarchy leaf node with WRED enabled as its congestion management mode has zero or one private WRED context (only one leaf node using it) and/or zero, one or several shared WRED contexts (multiple leaf nodes use the same WRED context). A private WRED context is used to perform congestion management for a single leaf node, while a shared WRED context is used to perform congestion management for a group of leaf nodes.

The configuration of WRED private and shared contexts is done through the definition of WRED profiles. Any WRED profile can be used by one or several WRED contexts (either private or shared).

5.15.6 Packet Marking

The TM APIs have been provided to support various types of packet marking such as VLAN DEI packet marking (IEEE 802.1Q), IPv4/IPv6 ECN marking of TCP and SCTP packets (IETF RFC 3168) and IPv4/IPv6 DSCP packet marking (IETF RFC 2597). All VLAN frames of a given color get their DEI bit set if marking is enabled for this color. In case, when marking for a given color is not enabled, the DEI bit is left as is (either set or not).

All IPv4/IPv6 packets of a given color with ECN set to 2'b01 or 2'b10 carrying TCP or SCTP have their ECN set to 2'b11 if the marking feature is enabled for the current color, otherwise the ECN field is left as is.

All IPv4/IPv6 packets have their color marked into DSCP bits 3 and 4 as follows: green mapped to Low Drop Precedence (2'b01), yellow to Medium (2'b10) and red to High (2'b11). Marking needs to be explicitly enabled for each color; when not enabled for a given color, the DSCP field of all packets with that color is left as is.

5.15.7 Steps to Setup the Hierarchy

The TM hierarchical tree consists of leaf nodes and non-leaf nodes. Each leaf node sits on top of a scheduling queue of the current Ethernet port. Therefore, the leaf nodes have predefined IDs in the range of 0... (N-1), where N is the number of scheduling queues of the current Ethernet port. The non-leaf nodes have their IDs generated by the application outside of the above range, which is reserved for leaf nodes.

Each non-leaf node has multiple inputs (its children nodes) and single output (which is input to its parent node). It arbitrates its inputs using Strict Priority (SP) and Weighted Fair Queuing (WFQ) algorithms to schedule input packets to its output while observing its shaping (rate limiting) constraints.

The children nodes with different priorities are scheduled using the SP algorithm based on their priority, with 0 as the highest priority. Children with the same priority are scheduled using the WFQ algorithm according to their weights. The WFQ weight of a given child node is relative to the sum of the weights of all its sibling nodes that have the same priority, with 1 as the lowest weight. For each SP priority, the WFQ weight mode can be set as either byte-based or packet-based.

Initial Hierarchy Specification

The hierarchy is specified by incrementally adding nodes to build up the scheduling tree. The first node that is added to the hierarchy becomes the root node and all the nodes that are subsequently added have to be added as descendants of the root node. The parent of the root node has to be specified as RTE_TM_NODE_ID_NULL and there can only be one node with this parent ID (i.e. the root node). The unique ID that is assigned to each node when the node is created is further used to update the node configuration or to connect children nodes to it.

During this phase, some limited checks on the hierarchy specification can be conducted, usually limited in scope to the current node, its parent node and its sibling nodes. At this time, since the hierarchy is not fully defined, there is typically no real action performed by the underlying implementation.

Hierarchy Commit

The hierarchy commit API is called during the port initialization phase (before the Ethernet port is started) to freeze the start-up hierarchy. This function typically performs the following steps:

- It validates the start-up hierarchy that was previously defined for the current port through successive node add API invocations.
- Assuming successful validation, it performs all the necessary implementation specific operations to install the specified hierarchy on the current port, with immediate effect once the port is started.

This function fails when the currently configured hierarchy is not supported by the Ethernet port, in which case the user can abort or try out another hierarchy configuration (e.g. a hierarchy with less leaf nodes), which can be built from scratch or by modifying the existing hierarchy configuration. Note that this function can still fail due to other causes (e.g. not enough memory available in the system, etc.), even though the specified hierarchy is supported in principle by the current port.

Run-Time Hierarchy Updates

The TM API provides support for on-the-fly changes to the scheduling hierarchy, thus operations such as node add/delete, node suspend/resume, parent node update, etc., can be invoked after the Ethernet port has been started, subject to the specific implementation supporting them. The set of dynamic updates supported by the implementation is advertised through the port capability set.

5.16 Wireless Baseband Device Library

The Wireless Baseband library provides a common programming framework that abstracts HW accelerators based on FPGA and/or Fixed Function Accelerators that assist with 3GPP Physical Layer processing. Furthermore, it decouples the application from the compute-intensive wireless functions by abstracting their optimized libraries to appear as virtual bbdev devices.

The functional scope of the BBDEV library are those functions in relation to the 3GPP Layer 1 signal processing (channel coding, modulation, ...).

The framework currently only supports Turbo Code FEC function.

5.16.1 Design Principles

The Wireless Baseband library follows the same ideology of DPDK's Ethernet Device and Crypto Device frameworks. Wireless Baseband provides a generic acceleration abstraction framework which supports both physical (hardware) and virtual (software) wireless acceleration functions.

5.16.2 Device Management

Device Creation

Physical bbdev devices are discovered during the PCI probe/enumeration of the EAL function which is executed at DPDK initialization, based on their PCI device identifier, each unique PCI BDF (bus/bridge, device, function).

Virtual devices can be created by two mechanisms, either using the EAL command line options or from within the application using an EAL API directly.

From the command line using the –vdev EAL option

```
--vdev 'baseband_turbo_sw,max_nb_queues=8,socket_id=0'
```

Or using the rte_vdev_init API within the application code.

```
rte_vdev_init("baseband_turbo_sw", "max_nb_queues=2,socket_id=0")
```

All virtual bbdev devices support the following initialization parameters:

- `max_nb_queues` - maximum number of queues supported by the device.
- `socket_id` - socket on which to allocate the device resources on.

Device Identification

Each device, whether virtual or physical is uniquely designated by two identifiers:

- A unique device index used to designate the bbdev device in all functions exported by the bbdev API.
- A device name used to designate the bbdev device in console messages, for administration or debugging purposes. For ease of use, the port name includes the port index.

Device Configuration

From the application point of view, each instance of a bbdev device consists of one or more queues identified by queue IDs. While different devices may have different capabilities (e.g. support different operation types), all queues on a device support identical configuration possibilities. A queue is configured for only one type of operation and is configured at initialization time. When an operation is enqueued to a specific queue ID, the result is dequeued from the same queue ID.

Configuration of a device has two different levels: configuration that applies to the whole device, and configuration that applies to a single queue.

Device configuration is applied with `rte_bbdev_setup_queues(dev_id,num_queues,socket_id)` and queue configuration is applied with `rte_bbdev_queue_configure(dev_id,queue_id,conf)`. Note that, although all queues on a device support same capabilities, they can be configured differently and will then behave differently. Devices supporting interrupts can enable them by using `rte_bbdev_intr_enable(dev_id)`.

The configuration of each bbdev device includes the following operations:

- Allocation of resources, including hardware resources if a physical device.
- Resetting the device into a well-known default state.
- Initialization of statistics counters.

The `rte_bbdev_setup_queues` API is used to setup queues for a bbdev device.

```
int rte_bbdev_setup_queues(uint16_t dev_id, uint16_t num_queues,
        int socket_id);
```

- `num_queues` argument identifies the total number of queues to setup for this device.
- `socket_id` specifies which socket will be used to allocate the memory.

The `rte_bbdev_intr_enable` API is used to enable interrupts for a bbdev device, if supported by the driver. Should be called before starting the device.

```
int rte_bbdev_intr_enable(uint16_t dev_id);
```

Queues Configuration

Each bbdev devices queue is individually configured through the `rte_bbdev_queue_configure()` API. Each queue resources may be allocated on a specified socket.

```
struct rte_bbdev_queue_conf {
    int socket;
    uint32_t queue_size;
    uint8_t priority;
    bool deferred_start;
    enum rte_bbdev_op_type op_type;
};
```

Device & Queues Management

After initialization, devices are in a stopped state, so must be started by the application. If an application is finished using a device it can close the device. Once closed, it cannot be restarted.

```
int rte_bbdev_start(uint16_t dev_id)
int rte_bbdev_stop(uint16_t dev_id)
int rte_bbdev_close(uint16_t dev_id)
int rte_bbdev_queue_start(uint16_t dev_id, uint16_t queue_id)
int rte_bbdev_queue_stop(uint16_t dev_id, uint16_t queue_id)
```

By default, all queues are started when the device is started, but they can be stopped individually.

```
int rte_bbdev_queue_start(uint16_t dev_id, uint16_t queue_id)
int rte_bbdev_queue_stop(uint16_t dev_id, uint16_t queue_id)
```

Logical Cores, Memory and Queues Relationships

The bbdev poll mode device driver library supports NUMA architecture, in which a processor's logical cores and interfaces utilize it's local memory. Therefore with baseband operations, the mbuf being operated on should be allocated from memory pools created in the local memory. The buffers should, if possible, remain on the local processor to obtain the best performance results and buffer descriptors should be populated with mbufs allocated from a mempool allocated from local memory.

The run-to-completion model also performs better, especially in the case of virtual bbdev devices, if the baseband operation and data buffers are in local memory instead of a remote processor's memory. This is also true for the pipe-line model provided all logical cores used are located on the same processor.

Multiple logical cores should never share the same queue for enqueuing operations or dequeuing operations on the same bbdev device since this would require global locks and hinder performance. It is however possible to use a different logical core to dequeue an operation on a queue pair from the logical core which it was enqueued on. This means that a baseband burst enqueue/dequeue APIs are a logical place to transition from one logical core to another in a packet processing pipeline.

5.16.3 Device Operation Capabilities

Capabilities (in terms of operations supported, max number of queues, etc.) identify what a bbdev is capable of performing that differs from one device to another. For the full scope of the bbdev capability see the definition of the structure in the *DPDK API Reference*.

```
struct rte_bbdev_op_cap;
```

A device reports its capabilities when registering itself in the bbdev framework. With the aid of this capabilities mechanism, an application can query devices to discover which operations within the 3GPP physical layer they are capable of performing. Below is an example of the capabilities for a PMD it supports in relation to Turbo Encoding and Decoding operations.

```
static const struct rte_bbdev_op_cap bbdev_capabilities[] = {
    {
        .type = RTE_BBDEV_OP_TURBO_DEC,
        .cap.turbo_dec = {
            .capability_flags =
                RTE_BBDEV_TURBO_SUBBLOCK_DEINTERLEAVE |
```

(continues on next page)

(continued from previous page)

```
                RTE_BBDEV_TURBO_POS_LLR_1_BIT_IN |
                RTE_BBDEV_TURBO_NEG_LLR_1_BIT_IN |
                RTE_BBDEV_TURBO_CRC_TYPE_24B |
                RTE_BBDEV_TURBO_DEC_TB_CRC_24B_KEEP |
                RTE_BBDEV_TURBO_EARLY_TERMINATION,
            .max_llr_modulus = 16,
            .num_buffers_src = RTE_BBDEV_TURBO_MAX_CODE_BLOCKS,
            .num_buffers_hard_out =
                    RTE_BBDEV_TURBO_MAX_CODE_BLOCKS,
            .num_buffers_soft_out = 0,
        }
    },
    {
        .type   = RTE_BBDEV_OP_TURBO_ENC,
        .cap.turbo_enc = {
            .capability_flags =
                    RTE_BBDEV_TURBO_CRC_24B_ATTACH |
                    RTE_BBDEV_TURBO_CRC_24A_ATTACH |
                    RTE_BBDEV_TURBO_RATE_MATCH |
                    RTE_BBDEV_TURBO_RV_INDEX_BYPASS,
            .num_buffers_src = RTE_BBDEV_TURBO_MAX_CODE_BLOCKS,
            .num_buffers_dst = RTE_BBDEV_TURBO_MAX_CODE_BLOCKS,
        }
    },
    RTE_BBDEV_END_OF_CAPABILITIES_LIST()
};
```

Capabilities Discovery

Discovering the features and capabilities of a bbdev device poll mode driver is achieved through the `rte_bbdev_info_get()` function.

```
int rte_bbdev_info_get(uint16_t dev_id, struct rte_bbdev_info *dev_info)
```

This allows the user to query a specific bbdev PMD and get all the device capabilities. The `rte_bbdev_info` structure provides two levels of information:

- Device relevant information, like: name and related rte_bus.
- Driver specific information, as defined by the `struct rte_bbdev_driver_info` structure, this is where capabilities reside along with other specifics like: maximum queue sizes and priority level.

```
struct rte_bbdev_info {
    int socket_id;
    const char *dev_name;
    const struct rte_device *device;
    uint16_t num_queues;
    bool started;
    struct rte_bbdev_driver_info drv;
};
```

5.16.4 Operation Processing

Scheduling of baseband operations on DPDK's application data path is performed using a burst oriented asynchronous API set. A queue on a bbdev device accepts a burst of baseband operations using enqueue burst API. On physical bbdev devices the enqueue burst API will place the operations to be processed on the device's hardware input queue, for virtual devices the processing of the baseband operations is usually completed during the enqueue call to the bbdev device. The dequeue burst API will retrieve any processed operations available from the queue on the bbdev device, from physical devices this is usually directly from the device's processed queue, and for virtual device's from a `rte_ring` where processed operations are placed after being processed on the enqueue call.

Enqueue / Dequeue Burst APIs

The burst enqueue API uses a bbdev device identifier and a queue identifier to specify the bbdev device queue to schedule the processing on. The `num_ops` parameter is the number of operations to process which are supplied in the `ops` array of `rte_bbdev_*_op` structures. The enqueue function returns the number of operations it actually enqueued for processing, a return value equal to `num_ops` means that all packets have been enqueued.

```
uint16_t rte_bbdev_enqueue_enc_ops(uint16_t dev_id, uint16_t queue_id,
        struct rte_bbdev_enc_op **ops, uint16_t num_ops)

uint16_t rte_bbdev_enqueue_dec_ops(uint16_t dev_id, uint16_t queue_id,
        struct rte_bbdev_dec_op **ops, uint16_t num_ops)
```

The dequeue API uses the same format as the enqueue API of processed but the `num_ops` and `ops` parameters are now used to specify the max processed operations the user wishes to retrieve and the location in which to store them. The API call returns the actual number of processed operations returned, this can never be larger than `num_ops`.

```
uint16_t rte_bbdev_dequeue_enc_ops(uint16_t dev_id, uint16_t queue_id,
        struct rte_bbdev_enc_op **ops, uint16_t num_ops)

uint16_t rte_bbdev_dequeue_dec_ops(uint16_t dev_id, uint16_t queue_id,
        struct rte_bbdev_dec_op **ops, uint16_t num_ops)
```

Operation Representation

An encode bbdev operation is represented by `rte_bbdev_enc_op` structure, and by `rte_bbdev_dec_op` for decode. These structures act as metadata containers for all necessary information required for the bbdev operation to be processed on a particular bbdev device poll mode driver.

```
struct rte_bbdev_enc_op {
    int status;
    struct rte_mempool *mempool;
    void *opaque_data;
    union {
        struct rte_bbdev_op_turbo_enc turbo_enc;
        struct rte_bbdev_op_ldpc_enc ldpc_enc;
    }
};

struct rte_bbdev_dec_op {
    int status;
```

(continues on next page)

(continued from previous page)

```
    struct rte_mempool *mempool;
    void *opaque_data;
    union {
        struct rte_bbdev_op_turbo_dec turbo_enc;
        struct rte_bbdev_op_ldpc_dec ldpc_enc;
    }
};
```

The operation structure by itself defines the operation type. It includes an operation status, a reference to the operation specific data, which can vary in size and content depending on the operation being provisioned. It also contains the source mempool for the operation, if it is allocated from a mempool.

If bbdev operations are allocated from a bbdev operation mempool, see next section, there is also the ability to allocate private memory with the operation for applications purposes.

Application software is responsible for specifying all the operation specific fields in the `rte_bbdev_*_op` structure which are then used by the bbdev PMD to process the requested operation.

Operation Management and Allocation

The bbdev library provides an API set for managing bbdev operations which utilize the Mempool Library to allocate operation buffers. Therefore, it ensures that the bbdev operation is interleaved optimally across the channels and ranks for optimal processing.

```
struct rte_mempool *
rte_bbdev_op_pool_create(const char *name, enum rte_bbdev_op_type type,
        unsigned int num_elements, unsigned int cache_size,
        int socket_id)
```

`rte_bbdev_*_op_alloc_bulk()` and `rte_bbdev_*_op_free_bulk()` are used to allocate bbdev operations of a specific type from a given bbdev operation mempool.

```
int rte_bbdev_enc_op_alloc_bulk(struct rte_mempool *mempool,
        struct rte_bbdev_enc_op **ops, uint16_t num_ops)

int rte_bbdev_dec_op_alloc_bulk(struct rte_mempool *mempool,
        struct rte_bbdev_dec_op **ops, uint16_t num_ops)
```

`rte_bbdev_*_op_free_bulk()` is called by the application to return an operation to its allocating pool.

```
void rte_bbdev_dec_op_free_bulk(struct rte_bbdev_dec_op **ops,
        unsigned int num_ops)
void rte_bbdev_enc_op_free_bulk(struct rte_bbdev_enc_op **ops,
        unsigned int num_ops)
```

BBDEV Inbound/Outbound Memory

The bbdev operation structure contains all the mutable data relating to performing Turbo and LDPC coding on a referenced mbuf data buffer. It is used for either encode or decode operations.

Table 5.91: Operation I/O

FEC	In	Out
Turbo Encode	input	output
Turbo Decode	input	hard output
		soft output (optional)
LDPC Encode	input	output
LDPC Decode	input	hard output
	HQ combine (optional)	HQ combine (optional)
		soft output (optional)

It is expected that the application provides input and output mbuf pointers allocated and ready to use.

The baseband framework supports FEC coding on Code Blocks (CB) and Transport Blocks (TB).

For the output buffer(s), the application is required to provide an allocated and free mbuf, to which the resulting output will be written.

The support of split "scattered" buffers is a driver-specific feature, so it is reported individually by the supporting driver as a capability.

Input and output data buffers are identified by `rte_bbdev_op_data` structure, as follows:

```
struct rte_bbdev_op_data {
    struct rte_mbuf *data;
    uint32_t offset;
    uint32_t length;
};
```

This structure has three elements:

- `data`: This is the mbuf data structure representing the data for BBDEV operation.

  This mbuf pointer can point to one Code Block (CB) data buffer or multiple CBs contiguously located next to each other. A Transport Block (TB) represents a whole piece of data that is divided into one or more CBs. Maximum number of CBs can be contained in one TB is defined by `RTE_BBDEV_(TURBO/LDPC)MAX_CODE_BLOCKS`.

  An mbuf data structure cannot represent more than one TB. The smallest piece of data that can be contained in one mbuf is one CB. An mbuf can include one contiguous CB, subset of contiguous CBs that are belonging to one TB, or all contiguous CBs that belong to one TB.

  If a BBDEV PMD supports the extended capability "Scatter-Gather", then it is capable of collecting (gathering) non-contiguous (scattered) data from multiple locations in the memory. This capability is reported by the capability flags:

  - `RTE_BBDEV_TURBO_ENC_SCATTER_GATHER`, `RTE_BBDEV_TURBO_DEC_SCATTER_GATHER`,
  - `RTE_BBDEV_LDPC_ENC_SCATTER_GATHER`, `RTE_BBDEV_LDPC_DEC_SCATTER_GATHER`.

  Chained mbuf data structures are only accepted if a BBDEV PMD supports this feature. A chained mbuf can represent one non-contiguous CB or multiple non-contiguous CBs. The first mbuf seg-

ment in the given chained mbuf represents the first piece of the CB. Offset is only applicable to the first segment. `length` is the total length of the CB.

BBDEV driver is responsible for identifying where the split is and enqueue the split data to its internal queues.

If BBDEV PMD does not support this feature, it will assume inbound mbuf data contains one segment.

The output mbuf data though is always one segment, even if the input was a chained mbuf.

- `offset`: This is the starting point of the BBDEV (encode/decode) operation, in bytes.

  BBDEV starts to read data past this offset. In case of chained mbuf, this offset applies only to the first mbuf segment.

- `length`: This is the total data length to be processed in one operation, in bytes.

  In case the mbuf data is representing one CB, this is the length of the CB undergoing the operation. If it is for multiple CBs, this is the total length of those CBs undergoing the operation. If it is for one TB, this is the total length of the TB under operation. In case of chained mbuf, this data length includes the lengths of the "scattered" data segments undergoing the operation.

BBDEV Turbo Encode Operation

```
struct rte_bbdev_op_turbo_enc {
    struct rte_bbdev_op_data input;
    struct rte_bbdev_op_data output;

    uint32_t op_flags;
    uint8_t rv_index;
    uint8_t code_block_mode;
    union {
        struct rte_bbdev_op_enc_cb_params cb_params;
        struct rte_bbdev_op_enc_tb_params tb_params;
    };
};
```

The Turbo encode structure includes the `input` and `output` mbuf data pointers. The provided mbuf pointer of `input` needs to be big enough to stretch for extra CRC trailers.

Table 5.92: **struct rte_bbdev_op_turbo_enc** parameters

Parameter	Description
input	input CB or TB data
output	rate matched CB or TB output buffer
op_flags	bitmask of all active operation capabilities
rv_index	redundancy version index [0..3]
code_block_mode	code block or transport block mode
cb_params	code block specific parameters (code block mode only)
tb_params	transport block specific parameters (transport block mode only)

The encode interface works on both the code block (CB) and the transport block (TB). An operation executes in "CB-mode" when the CB is standalone. While "TB-mode" executes when an operation performs on one or multiple CBs that belong to a TB. Therefore, a given data can be standalone CB,

full-size TB or partial TB. Partial TB means that only a subset of CBs belonging to a bigger TB are being enqueued.

> **NOTE:** It is assumed that all enqueued ops in one `rte_bbdev_enqueue_enc_ops()` call belong to one mode, either CB-mode or TB-mode.

In case that the TB is smaller than Z (6144 bits), then effectively the TB = CB. CRC24A is appended to the tail of the CB. The application is responsible for calculating and appending CRC24A before calling BBDEV in case that the underlying driver does not support CRC24A generation.

In CB-mode, CRC24A/B is an optional operation. The CB parameter `k` is the size of the CB (this maps to K as described in 3GPP TS 36.212 section 5.1.2), this size is inclusive of CRC24A/B. The `length` is inclusive of CRC24A/B and equals to `k` in this case.

Not all BBDEV PMDs are capable of CRC24A/B calculation. Flags `RTE_BBDEV_TURBO_CRC_24A_ATTACH` and `RTE_BBDEV_TURBO_CRC_24B_ATTACH` informs the application with relevant capability. These flags can be set in the `op_flags` parameter to indicate to BBDEV to calculate and append CRC24A/B to CB before going forward with Turbo encoding.

Output format of the CB encode will have the encoded CB in `e` size output (this maps to E described in 3GPP TS 36.212 section 5.1.4.1.2). The output mbuf buffer size needs to be big enough to hold the encoded buffer of size `e`.

In TB-mode, CRC24A is assumed to be pre-calculated and appended to the inbound TB mbuf data buffer. The output mbuf data structure is expected to be allocated by the application with enough room for the output data.

The difference between the partial and full-size TB is that we need to know the index of the first CB in this group and the number of CBs contained within. The first CB index is given by `r` but the number of the remaining CBs is calculated automatically by BBDEV before passing down to the driver.

The number of remaining CBs should not be confused with `c`. `c` is the total number of CBs that composes the whole TB (this maps to C as described in 3GPP TS 36.212 section 5.1.2).

The `length` is total size of the CBs inclusive of any CRC24A and CRC24B in case they were appended by the application.

The case when one CB belongs to TB and is being enqueued individually to BBDEV, this case is considered as a special case of partial TB where its number of CBs is 1. Therefore, it requires to get processed in TB-mode.

The figure below visualizes the encoding of CBs using BBDEV interface in TB-mode. CB-mode is a reduced version, where only one CB exists:

Fig. 5.24: Turbo encoding of Code Blocks in mbuf structure

BBDEV Turbo Decode Operation

```
struct rte_bbdev_op_turbo_dec {
    struct rte_bbdev_op_data input;
    struct rte_bbdev_op_data hard_output;
    struct rte_bbdev_op_data soft_output;

    uint32_t op_flags;
    uint8_t rv_index;
    uint8_t iter_min:4;
    uint8_t iter_max:4;
    uint8_t iter_count;
    uint8_t ext_scale;
    uint8_t num_maps;
    uint8_t code_block_mode;
    union {
        struct rte_bbdev_op_dec_cb_params cb_params;
        struct rte_bbdev_op_dec_tb_params tb_params;
    };
};
```

The Turbo decode structure includes the `input`, `hard_output` and optionally the `soft_output` mbuf data pointers.

Table 5.93: **struct rte_bbdev_op_turbo_dec** parameters

Parameter	Description
input	virtual circular buffer, wk, size 3*Kpi for each CB
hard output	hard decisions buffer, decoded output, size K for each CB
soft output	soft LLR output buffer (optional)
op_flags	bitmask of all active operation capabilities
rv_index	redundancy version index [0..3]
iter_max	maximum number of iterations to perofrm in decode all CBs
iter_min	minimum number of iterations to perform in decoding all CBs
iter_count	number of iterations to performed in decoding all CBs
ext_scale	scale factor on extrinsic info (5 bits)
num_maps	number of MAP engines to use in decode
code_block_mode	code block or transport block mode
cb_params	code block specific parameters (code block mode only)
tb_params	transport block specific parameters (transport block mode only)

Similarly, the decode interface works on both the code block (CB) and the transport block (TB). An operation executes in "CB-mode" when the CB is standalone. While "TB-mode" executes when an operation performs on one or multiple CBs that belong to a TB. Therefore, a given data can be standalone CB, full-size TB or partial TB. Partial TB means that only a subset of CBs belonging to a bigger TB are being enqueued.

> **NOTE:** It is assumed that all enqueued ops in one `rte_bbdev_enqueue_dec_ops()` call belong to one mode, either CB-mode or TB-mode.

The CB parameter `k` is the size of the decoded CB (this maps to K as described in 3GPP TS 36.212 section 5.1.2), this size is inclusive of CRC24A/B. The `length` is inclusive of CRC24A/B and equals to `k` in this case.

The input encoded CB data is the Virtual Circular Buffer data stream, wk, with the null padding included as described in 3GPP TS 36.212 section 5.1.4.1.2 and shown in 3GPP TS 36.212 section 5.1.4.1 Figure

5.1.4-1. The size of the virtual circular buffer is 3*Kpi, where Kpi is the 32 byte aligned value of K, as specified in 3GPP TS 36.212 section 5.1.4.1.1.

Each byte in the input circular buffer is the LLR value of each bit of the original CB.

`hard_output` is a mandatory capability that all BBDEV PMDs support. This is the decoded CBs of K sizes (CRC24A/B is the last 24-bit in each decoded CB). Soft output is an optional capability for BBDEV PMDs. Setting flag `RTE_BBDEV_TURBO_DEC_TB_CRC_24B_KEEP` in `op_flags` directs BBDEV to retain CRC24B at the end of each CB. This might be useful for the application in debug mode. An LLR rate matched output is computed in the `soft_output` buffer structure for the given CB parameter `e` size (this maps to E described in 3GPP TS 36.212 section 5.1.4.1.2). The output mbuf buffer size needs to be big enough to hold the encoded buffer of size `e`.

The first CB Virtual Circular Buffer (VCB) index is given by `r` but the number of the remaining CB VCBs is calculated automatically by BBDEV before passing down to the driver.

The number of remaining CB VCBs should not be confused with `c`. `c` is the total number of CBs that composes the whole TB (this maps to C as described in 3GPP TS 36.212 section 5.1.2).

The `length` is total size of the CBs inclusive of any CRC24A and CRC24B in case they were appended by the application.

The case when one CB belongs to TB and is being enqueued individually to BBDEV, this case is considered as a special case of partial TB where its number of CBs is 1. Therefore, it requires to get processed in TB-mode.

The output mbuf data structure is expected to be allocated by the application with enough room for the output data.

The figure below visualizes the decoding of CBs using BBDEV interface in TB-mode. CB-mode is a reduced version, where only one CB exists:

Fig. 5.25: Turbo decoding of Code Blocks in mbuf structure

BBDEV LDPC Encode Operation

The operation flags that can be set for each LDPC encode operation are given below.

> **NOTE:** The actual operation flags that may be used with a specific BBDEV PMD are dependent on the driver capabilities as reported via `rte_bbdev_info_get()`, and may be a subset of those below.

Description of LDPC encode capability flags
RTE_BBDEV_LDPC_INTERLEAVER_BYPASS Set to bypass bit-level interleaver on output stream
RTE_BBDEV_LDPC_RATE_MATCH Set to enabling the RATE_MATCHING processing
RTE_BBDEV_LDPC_CRC_24A_ATTACH Set to attach transport block CRC-24A
RTE_BBDEV_LDPC_CRC_24B_ATTACH Set to attach code block CRC-24B
RTE_BBDEV_LDPC_CRC_16_ATTACH Set to attach code block CRC-16
RTE_BBDEV_LDPC_ENC_INTERRUPTS Set if a device supports encoder dequeue interrupts
RTE_BBDEV_LDPC_ENC_SCATTER_GATHER Set if a device supports scatter-gather functionality
RTE_BBDEV_LDPC_ENC_CONCATENATION Set if a device supports concatenation of non byte aligned output

The structure passed for each LDPC encode operation is given below, with the operation flags forming a bitmask in the `op_flags` field.

```
struct rte_bbdev_op_ldpc_enc {

    struct rte_bbdev_op_data input;
    struct rte_bbdev_op_data output;

    uint32_t op_flags;
    uint8_t rv_index;
    uint8_t basegraph;
    uint16_t z_c;
    uint16_t n_cb;
    uint8_t q_m;
    uint16_t n_filler;
    uint8_t code_block_mode;
    union {
        struct rte_bbdev_op_enc_ldpc_cb_params cb_params;
        struct rte_bbdev_op_enc_ldpc_tb_params tb_params;
    };
};
```

The LDPC encode parameters are set out in the table below.

Parameter	Description	
input	input CB or TB data	
output	rate matched CB or TB output buffer	
op_flags	bitmask of all active operation capabilities	
rv_index	redundancy version index [0..3]	
basegraph	Basegraph 1 or 2	
z_c	Zc, LDPC lifting size	
n_cb	Ncb, length of the circular buffer in bits.	
q_m	Qm, modulation order {2,4,6,8,10}	
n_filler	number of filler bits	
code_block_mode	code block or transport block mode	
op_flags	bitmask of all active operation capabilities	
cb_params	code block specific parameters (code block mode only)	
	e	E, length of the rate matched output sequence in bits
tb_params	transport block specific parameters (transport block mode only)	
	c	number of CBs in the TB or partial TB
	r	index of the first CB in the inbound mbuf data
	c_ab	number of CBs that use Ea before switching to Eb
	ea	Ea, length of the RM output sequence in bits, r < cab
	eb	Eb, length of the RM output sequence in bits, r >= cab

The mbuf input `input` is mandatory for all BBDEV PMDs and is the incoming code block or transport block data.

The mbuf output `output` is mandatory and is the encoded CB(s). In CB-mode ut contains the encoded CB of size `e` (E in 3GPP TS 38.212 section 6.2.5). In TB-mode it contains multiple contiguous encoded CBs of size `ea` or `eb`. The `output` buffer is allocated by the application with enough room for the output data.

The encode interface works on both a code block (CB) and a transport block (TB) basis.

> **NOTE:** All enqueued ops in one `rte_bbdev_enqueue_enc_ops()` call belong to one mode, either CB-mode or TB-mode.

The valid modes of operation are:

- CB-mode: one CB (attach CRC24B if required)
- CB-mode: one CB making up one TB (attach CRC24A if required)
- TB-mode: one or more CB of a partial TB (attach CRC24B(s) if required)
- TB-mode: one or more CB of a complete TB (attach CRC24AB(s) if required)

In CB-mode if `RTE_BBDEV_LDPC_CRC_24A_ATTACH` is set then CRC24A is appended to the CB. If `RTE_BBDEV_LDPC_CRC_24A_ATTACH` is not set the application is responsible for calculating and appending CRC24A before calling BBDEV. The input data mbuf `length` is inclusive of CRC24A/B where present and is equal to the code block size `K`.

In TB-mode, CRC24A is assumed to be pre-calculated and appended to the inbound TB data buffer,

unless the RTE_BBDEV_LDPC_CRC_24A_ATTACH flag is set when it is the responsibility of BBDEV. The input data mbuf `length` is total size of the CBs inclusive of any CRC24A and CRC24B in the case they were appended by the application.

Not all BBDEV PMDs may be capable of CRC24A/B calculation. Flags RTE_BBDEV_LDPC_CRC_24A_ATTACH and RTE_BBDEV_LDPC_CRC_24B_ATTACH inform the application of the relevant capability. These flags can be set in the `op_flags` parameter to indicate BBDEV to calculate and append CRC24A to CB before going forward with LDPC encoding.

The difference between the partial and full-size TB is that BBDEV needs the index of the first CB in this group and the number of CBs in the group. The first CB index is given by `r` but the number of the CBs is calculated by BBDEV before signalling to the driver.

The number of CBs in the group should not be confused with `c`, the total number of CBs in the full TB (`C` as per 3GPP TS 38.212 section 5.2.2)

Figure Fig. 5.24 above showing the Turbo encoding of CBs using BBDEV interface in TB-mode is also valid for LDPC encode.

BBDEV LDPC Decode Operation

The operation flags that can be set for each LDPC decode operation are given below.

> **NOTE:** The actual operation flags that may be used with a specific BBDEV PMD are dependent on the driver capabilities as reported via `rte_bbdev_info_get()`, and may be a subset of those below.

Description of LDPC decode capability flags
RTE_BBDEV_LDPC_CRC_TYPE_24A_CHECK Set for transport block CRC-24A checking
RTE_BBDEV_LDPC_CRC_TYPE_24B_CHECK Set for code block CRC-24B checking
RTE_BBDEV_LDPC_CRC_TYPE_24B_DROP Set to drop the last CRC bits decoding output
RTE_BBDEV_LDPC_DEINTERLEAVER_BYPASS Set for bit-level de-interleaver bypass on input stream
RTE_BBDEV_LDPC_HQ_COMBINE_IN_ENABLE Set for HARQ combined input stream enable
RTE_BBDEV_LDPC_HQ_COMBINE_OUT_ENABLE Set for HARQ combined output stream enable
RTE_BBDEV_LDPC_DECODE_BYPASS Set for LDPC decoder bypass RTE_BBDEV_LDPC_HQ_COMBINE_OUT_ENABLE must be set
RTE_BBDEV_LDPC_DECODE_SOFT_OUT Set for soft-output stream enable
RTE_BBDEV_LDPC_SOFT_OUT_RM_BYPASS Set for Rate-Matching bypass on soft-out stream
RTE_BBDEV_LDPC_SOFT_OUT_DEINTERLEAVER_BYPASS Set for bit-level de-interleaver bypass on soft-output stream
RTE_BBDEV_LDPC_ITERATION_STOP_ENABLE Set for iteration stopping on successful decode condition enable Where a successful decode is a successful syndrome check
RTE_BBDEV_LDPC_DEC_INTERRUPTS Set if a device supports decoder dequeue interrupts
RTE_BBDEV_LDPC_DEC_SCATTER_GATHER Set if a device supports scatter-gather functionality
RTE_BBDEV_LDPC_HARQ_6BIT_COMPRESSION Set if a device supports input/output HARQ compression

The structure passed for each LDPC decode operation is given below, with the operation flags forming a bitmask in the `op_flags` field.

```
struct rte_bbdev_op_ldpc_dec {

    struct rte_bbdev_op_data input;
    struct rte_bbdev_op_data hard_output;
    struct rte_bbdev_op_data soft_output;
    struct rte_bbdev_op_data harq_combined_input;
    struct rte_bbdev_op_data harq_combined_output;

    uint32_t op_flags;
    uint8_t rv_index;
    uint8_t basegraph;
    uint16_t z_c;
    uint16_t n_cb;
    uint8_t q_m;
    uint16_t n_filler;
    uint8_t iter_max;
    uint8_t iter_count;
    uint8_t code_block_mode;
    union {
        struct rte_bbdev_op_dec_ldpc_cb_params cb_params;
        struct rte_bbdev_op_dec_ldpc_tb_params tb_params;
    };
};
```

The LDPC decode parameters are set out in the table below.

Parameter	Description	
input	input CB or TB data	
hard_output	hard decisions buffer, decoded output	
soft_output	soft LLR output buffer (optional)	
harq_comb_input	HARQ combined input buffer (optional)	
harq_comb_output	HARQ combined output buffer (optional)	
op_flags	bitmask of all active operation capabilities	
rv_index	redundancy version index [0..3]	
basegraph	Basegraph 1 or 2	
z_c	Zc, LDPC lifting size	
n_cb	Ncb, length of the circular buffer in bits.	
q_m	Qm, modulation order {1,2,4,6,8} from pi/2-BPSK to 256QAM	
n_filler	number of filler bits	
iter_max	maximum number of iterations to perform in decode all CBs	
iter_count	number of iterations performed in decoding all CBs	
code_block_mode	code block or transport block mode	
op_flags	bitmask of all active operation capabilities	
cb_params	code block specific parameters (code block mode only)	
	e	E, length of the rate matched output sequence in bits
tb_params	transport block specific parameters (transport block mode only)	
	c	number of CBs in the TB or partial TB
	r	index of the first CB in the inbound mbuf data
	c_ab	number of CBs that use Ea before switching to Eb
	ea	Ea, length of the RM output sequence in bits, r < cab
	eb	Eb, length of the RM output sequence in bits r >= cab

The mbuf input `input` encoded CB data is mandatory for all BBDEV PMDs and is the Virtual Circular Buffer data stream with null padding. Each byte in the input circular buffer is the LLR value of each bit of the original CB.

The mbuf output `hard_output` is mandatory and is the decoded CBs size K (CRC24A/B is the last 24-bit in each decoded CB).

The mbuf output `soft_output` is optional and is an LLR rate matched output of size `e` (this is `E` as per 3GPP TS 38.212 section 6.2.5).

The mbuf input `harq_combine_input` is optional and is a buffer with the input to the HARQ combination function of the device. If the capability RTE_BBDEV_LDPC_INTERNAL_HARQ_MEMORY_IN_ENABLE is set then the HARQ is stored in memory internal to the device and not visible to BBDEV.

The mbuf output `harq_combine_output` is optional and is a buffer for the output of the HARQ combination function of the device. If the capability RTE_BBDEV_LDPC_INTERNAL_HARQ_MEMORY_OUT_ENABLE is set then the HARQ is stored in memory internal to the device and not visible to BBDEV.

The output mbuf data structures are expected to be allocated by the application with enough room for the output data.

As with the LDPC encode, the decode interface works on both a code block (CB) and a transport block

(TB) basis.

> **NOTE:** All enqueued ops in one `rte_bbdev_enqueue_dec_ops()` call belong to one mode, either CB-mode or TB-mode.

The valid modes of operation are:

- CB-mode: one CB (check CRC24B if required)
- CB-mode: one CB making up one TB (check CRC24A if required)
- TB-mode: one or more CB making up a partial TB (check CRC24B(s) if required)
- TB-mode: one or more CB making up a complete TB (check CRC24B(s) if required)

The mbuf `length` is inclusive of CRC24A/B where present and is equal the code block size `K`.

The first CB Virtual Circular Buffer (VCB) index is given by `r` but the number of the remaining CB VCBs is calculated automatically by BBDEV and passed down to the driver.

The number of remaining CB VCBs should not be confused with `c`, the total number of CBs in the full TB (`C` as per 3GPP TS 38.212 section 5.2.2)

The `length` is total size of the CBs inclusive of any CRC24A and CRC24B in case they were appended by the application.

Figure Fig. 5.25 above showing the Turbo decoding of CBs using BBDEV interface in TB-mode is also valid for LDPC decode.

5.16.5 Sample code

The baseband device sample application gives an introduction on how to use the bbdev framework, by giving a sample code performing a loop-back operation with a baseband processor capable of transceiving data packets.

The following sample C-like pseudo-code shows the basic steps to encode several buffers using (**sw_turbo**) bbdev PMD.

```
/* EAL Init */
ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");

/* Get number of available bbdev devices */
nb_bbdevs = rte_bbdev_count();
if (nb_bbdevs == 0)
    rte_exit(EXIT_FAILURE, "No bbdevs detected!\n");

/* Create bbdev op pools */
bbdev_op_pool[RTE_BBDEV_OP_TURBO_ENC] =
        rte_bbdev_op_pool_create("bbdev_op_pool_enc",
        RTE_BBDEV_OP_TURBO_ENC, NB_MBUF, 128, rte_socket_id());

/* Get information for this device */
rte_bbdev_info_get(dev_id, &info);

/* Setup BBDEV device queues */
ret = rte_bbdev_setup_queues(dev_id, qs_nb, info.socket_id);
if (ret < 0)
    rte_exit(EXIT_FAILURE,
```

(continues on next page)

(continued from previous page)

```
            "ERROR(%d): BBDEV %u not configured properly\n",
            ret, dev_id);

/* setup device queues */
qconf.socket = info.socket_id;
qconf.queue_size = info.drv.queue_size_lim;
qconf.op_type = RTE_BBDEV_OP_TURBO_ENC;

for (q_id = 0; q_id < qs_nb; q_id++) {
    /* Configure all queues belonging to this bbdev device */
    ret = rte_bbdev_queue_configure(dev_id, q_id, &qconf);
    if (ret < 0)
        rte_exit(EXIT_FAILURE,
                "ERROR(%d): BBDEV %u queue %u not configured properly\n",
                ret, dev_id, q_id);
}

/* Start bbdev device */
ret = rte_bbdev_start(dev_id);

/* Create the mbuf mempool for pkts */
mbuf_pool = rte_pktmbuf_pool_create("bbdev_mbuf_pool",
        NB_MBUF, MEMPOOL_CACHE_SIZE, 0,
        RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());
if (mbuf_pool == NULL)
    rte_exit(EXIT_FAILURE,
            "Unable to create '%s' pool\n", pool_name);

while (!global_exit_flag) {

    /* Allocate burst of op structures in preparation for enqueue */
    if (rte_bbdev_enc_op_alloc_bulk(bbdev_op_pool[RTE_BBDEV_OP_TURBO_ENC],
        ops_burst, op_num) != 0)
        continue;

    /* Allocate input mbuf pkts */
    ret = rte_pktmbuf_alloc_bulk(mbuf_pool, input_pkts_burst, MAX_PKT_BURST);
    if (ret < 0)
        continue;

    /* Allocate output mbuf pkts */
    ret = rte_pktmbuf_alloc_bulk(mbuf_pool, output_pkts_burst, MAX_PKT_BURST);
    if (ret < 0)
        continue;

    for (j = 0; j < op_num; j++) {
        /* Append the size of the ethernet header */
        rte_pktmbuf_append(input_pkts_burst[j],
                sizeof(struct rte_ether_hdr));

        /* set op */

        ops_burst[j]->turbo_enc.input.offset =
            sizeof(struct rte_ether_hdr);

        ops_burst[j]->turbo_enc->input.length =
            rte_pktmbuf_pkt_len(bbdev_pkts[j]);

        ops_burst[j]->turbo_enc->input.data =
            input_pkts_burst[j];
```

(continues on next page)

(continued from previous page)

```
        ops_burst[j]->turbo_enc->output.offset =
            sizeof(struct rte_ether_hdr);

        ops_burst[j]->turbo_enc->output.data =
                output_pkts_burst[j];
    }

    /* Enqueue packets on BBDEV device */
    op_num = rte_bbdev_enqueue_enc_ops(qconf->bbdev_id,
            qconf->bbdev_qs[q], ops_burst,
            MAX_PKT_BURST);

    /* Dequeue packets from BBDEV device*/
    op_num = rte_bbdev_dequeue_enc_ops(qconf->bbdev_id,
            qconf->bbdev_qs[q], ops_burst,
            MAX_PKT_BURST);
}
```

BBDEV Device API

The bbdev Library API is described in the *DPDK API Reference* document.

5.17 Cryptography Device Library

The cryptodev library provides a Crypto device framework for management and provisioning of hardware and software Crypto poll mode drivers, defining generic APIs which support a number of different Crypto operations. The framework currently only supports cipher, authentication, chained cipher/authentication and AEAD symmetric and asymmetric Crypto operations.

5.17.1 Design Principles

The cryptodev library follows the same basic principles as those used in DPDK's Ethernet Device framework. The Crypto framework provides a generic Crypto device framework which supports both physical (hardware) and virtual (software) Crypto devices as well as a generic Crypto API which allows Crypto devices to be managed and configured and supports Crypto operations to be provisioned on Crypto poll mode driver.

5.17.2 Device Management

Device Creation

Physical Crypto devices are discovered during the PCI probe/enumeration of the EAL function which is executed at DPDK initialization, based on their PCI device identifier, each unique PCI BDF (bus/bridge, device, function). Specific physical Crypto devices, like other physical devices in DPDK can be white-listed or black-listed using the EAL command line options.

Virtual devices can be created by two mechanisms, either using the EAL command line options or from within the application using an EAL API directly.

From the command line using the –vdev EAL option

```
--vdev  'crypto_aesni_mb0,max_nb_queue_pairs=2,socket_id=0'
```

Note:

- If DPDK application requires multiple software crypto PMD devices then required number of `--vdev` with appropriate libraries are to be added.
- An Application with crypto PMD instances sharing the same library requires unique ID.

Example: `--vdev 'crypto_aesni_mb0' --vdev 'crypto_aesni_mb1'`

Or using the rte_vdev_init API within the application code.

```
rte_vdev_init("crypto_aesni_mb",
                  "max_nb_queue_pairs=2,socket_id=0")
```

All virtual Crypto devices support the following initialization parameters:

- `max_nb_queue_pairs` - maximum number of queue pairs supported by the device.
- `socket_id` - socket on which to allocate the device resources on.

Device Identification

Each device, whether virtual or physical is uniquely designated by two identifiers:

- A unique device index used to designate the Crypto device in all functions exported by the cryptodev API.
- A device name used to designate the Crypto device in console messages, for administration or debugging purposes. For ease of use, the port name includes the port index.

Device Configuration

The configuration of each Crypto device includes the following operations:

- Allocation of resources, including hardware resources if a physical device.
- Resetting the device into a well-known default state.
- Initialization of statistics counters.

The rte_cryptodev_configure API is used to configure a Crypto device.

```
int rte_cryptodev_configure(uint8_t dev_id,
                            struct rte_cryptodev_config *config)
```

The `rte_cryptodev_config` structure is used to pass the configuration parameters for socket selection and number of queue pairs.

```
struct rte_cryptodev_config {
    int socket_id;
    /**< Socket to allocate resources on */
    uint16_t nb_queue_pairs;
    /**< Number of queue pairs to configure on device */
};
```

Configuration of Queue Pairs

Each Crypto devices queue pair is individually configured through the `rte_cryptodev_queue_pair_setup` API. Each queue pairs resources may be allocated on a specified socket.

```
int rte_cryptodev_queue_pair_setup(uint8_t dev_id, uint16_t queue_pair_id,
            const struct rte_cryptodev_qp_conf *qp_conf,
            int socket_id)

struct rte_cryptodev_qp_conf {
    uint32_t nb_descriptors; /**< Number of descriptors per queue pair */
    struct rte_mempool *mp_session;
    /**< The mempool for creating session in sessionless mode */
    struct rte_mempool *mp_session_private;
    /**< The mempool for creating sess private data in sessionless mode */
};
```

The fields `mp_session` and `mp_session_private` are used for creating temporary session to process the crypto operations in the session-less mode. They can be the same other different mempools. Please note not all Cryptodev PMDs supports session-less mode.

Logical Cores, Memory and Queues Pair Relationships

The Crypto device Library as the Poll Mode Driver library support NUMA for when a processor's logical cores and interfaces utilize its local memory. Therefore Crypto operations, and in the case of symmetric Crypto operations, the session and the mbuf being operated on, should be allocated from memory pools created in the local memory. The buffers should, if possible, remain on the local processor to obtain the best performance results and buffer descriptors should be populated with mbufs allocated from a mempool allocated from local memory.

The run-to-completion model also performs better, especially in the case of virtual Crypto devices, if the Crypto operation and session and data buffer is in local memory instead of a remote processor's memory. This is also true for the pipe-line model provided all logical cores used are located on the same processor.

Multiple logical cores should never share the same queue pair for enqueuing operations or dequeuing operations on the same Crypto device since this would require global locks and hinder performance. It is however possible to use a different logical core to dequeue an operation on a queue pair from the logical core which it was enqueued on. This means that a crypto burst enqueue/dequeue APIs are a logical place to transition from one logical core to another in a packet processing pipeline.

5.17.3 Device Features and Capabilities

Crypto devices define their functionality through two mechanisms, global device features and algorithm capabilities. Global devices features identify device wide level features which are applicable to the whole device such as the device having hardware acceleration or supporting symmetric and/or asymmetric Crypto operations.

The capabilities mechanism defines the individual algorithms/functions which the device supports, such as a specific symmetric Crypto cipher, authentication operation or Authenticated Encryption with Associated Data (AEAD) operation.

Device Features

Currently the following Crypto device features are defined:

- Symmetric Crypto operations
- Asymmetric Crypto operations
- Chaining of symmetric Crypto operations
- SSE accelerated SIMD vector operations
- AVX accelerated SIMD vector operations
- AVX2 accelerated SIMD vector operations
- AESNI accelerated instructions
- Hardware off-load processing

Device Operation Capabilities

Crypto capabilities which identify particular algorithm which the Crypto PMD supports are defined by the operation type, the operation transform, the transform identifier and then the particulars of the transform. For the full scope of the Crypto capability see the definition of the structure in the *DPDK API Reference*.

```
struct rte_cryptodev_capabilities;
```

Each Crypto poll mode driver defines its own private array of capabilities for the operations it supports. Below is an example of the capabilities for a PMD which supports the authentication algorithm SHA1_HMAC and the cipher algorithm AES_CBC.

```
static const struct rte_cryptodev_capabilities pmd_capabilities[] = {
    {    /* SHA1 HMAC */
        .op = RTE_CRYPTO_OP_TYPE_SYMMETRIC,
        .sym = {
            .xform_type = RTE_CRYPTO_SYM_XFORM_AUTH,
            .auth = {
                .algo = RTE_CRYPTO_AUTH_SHA1_HMAC,
                .block_size = 64,
                .key_size = {
                    .min = 64,
                    .max = 64,
                    .increment = 0
                },
                .digest_size = {
                    .min = 12,
                    .max = 12,
                    .increment = 0
                },
                .aad_size = { 0 },
                .iv_size = { 0 }
            }
        }
    },
    {    /* AES CBC */
        .op = RTE_CRYPTO_OP_TYPE_SYMMETRIC,
        .sym = {
            .xform_type = RTE_CRYPTO_SYM_XFORM_CIPHER,
```

(continues on next page)

(continued from previous page)

```
            .cipher = {
                .algo = RTE_CRYPTO_CIPHER_AES_CBC,
                .block_size = 16,
                .key_size = {
                    .min = 16,
                    .max = 32,
                    .increment = 8
                },
                .iv_size = {
                    .min = 16,
                    .max = 16,
                    .increment = 0
                }
            }
        }
    }
}
```

Capabilities Discovery

Discovering the features and capabilities of a Crypto device poll mode driver is achieved through the `rte_cryptodev_info_get` function.

```
void rte_cryptodev_info_get(uint8_t dev_id,
                            struct rte_cryptodev_info *dev_info);
```

This allows the user to query a specific Crypto PMD and get all the device features and capabilities. The `rte_cryptodev_info` structure contains all the relevant information for the device.

```
struct rte_cryptodev_info {
    const char *driver_name;
    uint8_t driver_id;
    struct rte_device *device;

    uint64_t feature_flags;

    const struct rte_cryptodev_capabilities *capabilities;

    unsigned max_nb_queue_pairs;

    struct {
        unsigned max_nb_sessions;
    } sym;
};
```

5.17.4 Operation Processing

Scheduling of Crypto operations on DPDK's application data path is performed using a burst oriented asynchronous API set. A queue pair on a Crypto device accepts a burst of Crypto operations using enqueue burst API. On physical Crypto devices the enqueue burst API will place the operations to be processed on the devices hardware input queue, for virtual devices the processing of the Crypto operations is usually completed during the enqueue call to the Crypto device. The dequeue burst API will retrieve any processed operations available from the queue pair on the Crypto device, from physical devices this is usually directly from the devices processed queue, and for virtual device's from a `rte_ring` where processed operations are placed after being processed on the enqueue call.

Private data

For session-based operations, the set and get API provides a mechanism for an application to store and retrieve the private user data information stored along with the crypto session.

For example, suppose an application is submitting a crypto operation with a session associated and wants to indicate private user data information which is required to be used after completion of the crypto operation. In this case, the application can use the set API to set the user data and retrieve it using get API.

```
int rte_cryptodev_sym_session_set_user_data(
        struct rte_cryptodev_sym_session *sess, void *data, uint16_t size);

void * rte_cryptodev_sym_session_get_user_data(
        struct rte_cryptodev_sym_session *sess);
```

Please note the `size` passed to set API cannot be bigger than the predefined `user_data_sz` when creating the session header mempool, otherwise the function will return error. Also when `user_data_sz` was defined as `0` when creating the session header mempool, the get API will always return `NULL`.

For session-less mode, the private user data information can be placed along with the `struct rte_crypto_op`. The `rte_crypto_op::private_data_offset` indicates the start of private data information. The offset is counted from the start of the rte_crypto_op including other crypto information such as the IVs (since there can be an IV also for authentication).

Enqueue / Dequeue Burst APIs

The burst enqueue API uses a Crypto device identifier and a queue pair identifier to specify the Crypto device queue pair to schedule the processing on. The `nb_ops` parameter is the number of operations to process which are supplied in the `ops` array of `rte_crypto_op` structures. The enqueue function returns the number of operations it actually enqueued for processing, a return value equal to `nb_ops` means that all packets have been enqueued.

```
uint16_t rte_cryptodev_enqueue_burst(uint8_t dev_id, uint16_t qp_id,
                                     struct rte_crypto_op **ops, uint16_t nb_ops)
```

The dequeue API uses the same format as the enqueue API of processed but the `nb_ops` and `ops` parameters are now used to specify the max processed operations the user wishes to retrieve and the location in which to store them. The API call returns the actual number of processed operations returned, this can never be larger than `nb_ops`.

```
uint16_t rte_cryptodev_dequeue_burst(uint8_t dev_id, uint16_t qp_id,
                                     struct rte_crypto_op **ops, uint16_t nb_ops)
```

Operation Representation

An Crypto operation is represented by an rte_crypto_op structure, which is a generic metadata container for all necessary information required for the Crypto operation to be processed on a particular Crypto device poll mode driver.

The operation structure includes the operation type, the operation status and the session type (session-based/less), a reference to the operation specific data, which can vary in size and content depending on the operation being provisioned. It also contains the source mempool for the operation, if it allocated from a mempool.

If Crypto operations are allocated from a Crypto operation mempool, see next section, there is also the ability to allocate private memory with the operation for applications purposes.

Application software is responsible for specifying all the operation specific fields in the `rte_crypto_op` structure which are then used by the Crypto PMD to process the requested operation.

Operation Management and Allocation

The cryptodev library provides an API set for managing Crypto operations which utilize the Mempool Library to allocate operation buffers. Therefore, it ensures that the crypto operation is interleaved optimally across the channels and ranks for optimal processing. A `rte_crypto_op` contains a field indicating the pool that it originated from. When calling `rte_crypto_op_free(op)`, the operation returns to its original pool.

```
extern struct rte_mempool *
rte_crypto_op_pool_create(const char *name, enum rte_crypto_op_type type,
                          unsigned nb_elts, unsigned cache_size, uint16_t priv_size,
                          int socket_id);
```

During pool creation `rte_crypto_op_init()` is called as a constructor to initialize each Crypto operation which subsequently calls `__rte_crypto_op_reset()` to configure any operation type specific fields based on the type parameter.

`rte_crypto_op_alloc()` and `rte_crypto_op_bulk_alloc()` are used to allocate Crypto operations of a specific type from a given Crypto operation mempool. `__rte_crypto_op_reset()` is called on each operation before being returned to allocate to a user so the operation is always in a good known state before use by the application.

```
struct rte_crypto_op *rte_crypto_op_alloc(struct rte_mempool *mempool,
                                          enum rte_crypto_op_type type)

unsigned rte_crypto_op_bulk_alloc(struct rte_mempool *mempool,
                                  enum rte_crypto_op_type type,
                                  struct rte_crypto_op **ops, uint16_t nb_ops)
```

`rte_crypto_op_free()` is called by the application to return an operation to its allocating pool.

```
void rte_crypto_op_free(struct rte_crypto_op *op)
```

5.17.5 Symmetric Cryptography Support

The cryptodev library currently provides support for the following symmetric Crypto operations; cipher, authentication, including chaining of these operations, as well as also supporting AEAD operations.

Session and Session Management

Sessions are used in symmetric cryptographic processing to store the immutable data defined in a cryptographic transform which is used in the operation processing of a packet flow. Sessions are used to manage information such as expand cipher keys and HMAC IPADs and OPADs, which need to be calculated for a particular Crypto operation, but are immutable on a packet to packet basis for a flow. Crypto sessions cache this immutable data in a optimal way for the underlying PMD and this allows further acceleration of the offload of Crypto workloads.

The Crypto device framework provides APIs to create session mempool and allocate and initialize sessions for crypto devices, where sessions are mempool objects. The application has to use `rte_cryptodev_sym_session_pool_create()` to create the session header mempool that creates a mempool with proper element size automatically and stores necessary information for safely accessing the session in the mempool's private data field.

To create a mempool for storing session private data, the application has two options. The first is to create another mempool with elt size equal to or bigger than the maximum session private data size of all crypto devices that will share the same session header. The creation of the mempool shall use the traditional `rte_mempool_create()` with the correct `elt_size`. The other option is to change the `elt_size` parameter in `rte_cryptodev_sym_session_pool_create()` to the correct value. The first option is more complex to implement but may result in better memory usage as a session header normally takes smaller memory footprint as the session private data.

Once the session mempools have been created, `rte_cryptodev_sym_session_create()` is used to allocate an uninitialized session from the given mempool. The session then must be initialized using `rte_cryptodev_sym_session_init()` for each of the required crypto devices. A symmetric transform chain is used to specify the operation and its parameters. See the section below for details on transforms.

When a session is no longer used, user must call `rte_cryptodev_sym_session_clear()` for each of the crypto devices that are using the session, to free all driver private session data. Once this is done, session should be freed using `rte_cryptodev_sym_session_free` which returns them to their mempool.

Transforms and Transform Chaining

Symmetric Crypto transforms (`rte_crypto_sym_xform`) are the mechanism used to specify the details of the Crypto operation. For chaining of symmetric operations such as cipher encrypt and authentication generate, the next pointer allows transform to be chained together. Crypto devices which support chaining must publish the chaining of symmetric Crypto operations feature flag. Allocation of the xform structure is in the application domain. To allow future API extensions in a backwardly compatible manner, e.g. addition of a new parameter, the application should zero the full xform struct before populating it.

Currently there are three transforms types cipher, authentication and AEAD. Also it is important to note that the order in which the transforms are passed indicates the order of the chaining.

```
struct rte_crypto_sym_xform {
    struct rte_crypto_sym_xform *next;
    /**< next xform in chain */
    enum rte_crypto_sym_xform_type type;
    /**< xform type */
    union {
        struct rte_crypto_auth_xform auth;
        /**< Authentication / hash xform */
        struct rte_crypto_cipher_xform cipher;
        /**< Cipher xform */
        struct rte_crypto_aead_xform aead;
        /**< AEAD xform */
    };
};
```

The API does not place a limit on the number of transforms that can be chained together but this will be limited by the underlying Crypto device poll mode driver which is processing the operation.

Symmetric Operations

The symmetric Crypto operation structure contains all the mutable data relating to performing symmetric cryptographic processing on a referenced mbuf data buffer. It is used for either cipher, authentication, AEAD and chained operations.

As a minimum the symmetric operation must have a source data buffer (`m_src`), a valid session (or transform chain if in session-less mode) and the minimum authentication/ cipher/ AEAD parameters required depending on the type of operation specified in the session or the transform chain.

```
struct rte_crypto_sym_op {
    struct rte_mbuf *m_src;
    struct rte_mbuf *m_dst;

    union {
        struct rte_cryptodev_sym_session *session;
        /**< Handle for the initialised session context */
        struct rte_crypto_sym_xform *xform;
        /**< Session-less API Crypto operation parameters */
    };

    union {
        struct {
            struct {
                uint32_t offset;
                uint32_t length;
            } data; /**< Data offsets and length for AEAD */

            struct {
                uint8_t *data;
                rte_iova_t phys_addr;
            } digest; /**< Digest parameters */

            struct {
                uint8_t *data;
                rte_iova_t phys_addr;
            } aad;
            /**< Additional authentication parameters */
        } aead;
```

(continues on next page)

(continued from previous page)

```
        struct {
            struct {
                struct {
                    uint32_t offset;
                    uint32_t length;
                } data; /**< Data offsets and length for ciphering */
            } cipher;

            struct {
                struct {
                    uint32_t offset;
                    uint32_t length;
                } data;
                /**< Data offsets and length for authentication */

                struct {
                    uint8_t *data;
                    rte_iova_t phys_addr;
                } digest; /**< Digest parameters */
            } auth;
        };
    };
};
```

5.17.6 Synchronous mode

Some cryptodevs support synchronous mode alongside with a standard asynchronous mode. In that case operations are performed directly when calling `rte_cryptodev_sym_cpu_crypto_process` method instead of enqueuing and dequeuing an operation before. This mode of operation allows cryptodevs which utilize CPU cryptographic acceleration to have significant performance boost comparing to standard asynchronous approach. Cryptodevs supporting synchronous mode have `RTE_CRYPTODEV_FF_SYM_CPU_CRYPTO` feature flag set.

To perform a synchronous operation a call to `rte_cryptodev_sym_cpu_crypto_process` has to be made with vectorized operation descriptor (`struct rte_crypto_sym_vec`) containing:

- `num` - number of operations to perform,
- pointer to an array of size `num` containing a scatter-gather list descriptors of performed operations (`struct rte_crypto_sgl`). Each instance of `struct rte_crypto_sgl` consists of a number of segments and a pointer to an array of segment descriptors `struct rte_crypto_vec`;
- pointers to arrays of size `num` containing IV, AAD and digest information,
- pointer to an array of size `num` where status information will be stored for each operation.

Function returns a number of successfully completed operations and sets appropriate status number for each operation in the status array provided as a call argument. Status different than zero must be treated as error.

For more details, e.g. how to convert an mbuf to an SGL, please refer to an example usage in the IPsec library implementation.

5.17.7 Sample code

There are various sample applications that show how to use the cryptodev library, such as the L2fwd with Crypto sample application (L2fwd-crypto) and the IPsec Security Gateway application (ipsec-secgw).

While these applications demonstrate how an application can be created to perform generic crypto operation, the required complexity hides the basic steps of how to use the cryptodev APIs.

The following sample code shows the basic steps to encrypt several buffers with AES-CBC (although performing other crypto operations is similar), using one of the crypto PMDs available in DPDK.

```
/*
 * Simple example to encrypt several buffers with AES-CBC using
 * the Cryptodev APIs.
 */

#define MAX_SESSIONS         1024
#define NUM_MBUFS            1024
#define POOL_CACHE_SIZE      128
#define BURST_SIZE           32
#define BUFFER_SIZE          1024
#define AES_CBC_IV_LENGTH    16
#define AES_CBC_KEY_LENGTH   16
#define IV_OFFSET            (sizeof(struct rte_crypto_op) + \
                             sizeof(struct rte_crypto_sym_op))

struct rte_mempool *mbuf_pool, *crypto_op_pool;
struct rte_mempool *session_pool, *session_priv_pool;
unsigned int session_size;
int ret;

/* Initialize EAL. */
ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");

uint8_t socket_id = rte_socket_id();

/* Create the mbuf pool. */
mbuf_pool = rte_pktmbuf_pool_create("mbuf_pool",
                                    NUM_MBUFS,
                                    POOL_CACHE_SIZE,
                                    0,
                                    RTE_MBUF_DEFAULT_BUF_SIZE,
                                    socket_id);
if (mbuf_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot create mbuf pool\n");

/*
 * The IV is always placed after the crypto operation,
 * so some private data is required to be reserved.
 */
unsigned int crypto_op_private_data = AES_CBC_IV_LENGTH;

/* Create crypto operation pool. */
crypto_op_pool = rte_crypto_op_pool_create("crypto_op_pool",
                                           RTE_CRYPTO_OP_TYPE_SYMMETRIC,
                                           NUM_MBUFS,
                                           POOL_CACHE_SIZE,
                                           crypto_op_private_data,
                                           socket_id);
```

(continues on next page)

(continued from previous page)

```
if (crypto_op_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot create crypto op pool\n");

/* Create the virtual crypto device. */
char args[128];
const char *crypto_name = "crypto_aesni_mb0";
snprintf(args, sizeof(args), "socket_id=%d", socket_id);
ret = rte_vdev_init(crypto_name, args);
if (ret != 0)
    rte_exit(EXIT_FAILURE, "Cannot create virtual device");

uint8_t cdev_id = rte_cryptodev_get_dev_id(crypto_name);

/* Get private session data size. */
session_size = rte_cryptodev_sym_get_private_session_size(cdev_id);

#ifdef USE_TWO_MEMPOOLS
/* Create session mempool for the session header. */
session_pool = rte_cryptodev_sym_session_pool_create("session_pool",
                                MAX_SESSIONS,
                                0,
                                POOL_CACHE_SIZE,
                                0,
                                socket_id);

/*
 * Create session private data mempool for the
 * private session data for the crypto device.
 */
session_priv_pool = rte_mempool_create("session_pool",
                                MAX_SESSIONS,
                                session_size,
                                POOL_CACHE_SIZE,
                                0, NULL, NULL, NULL,
                                NULL, socket_id,
                                0);

#else
/* Use of the same mempool for session header and private data */
    session_pool = rte_cryptodev_sym_session_pool_create("session_pool",
                                MAX_SESSIONS * 2,
                                session_size,
                                POOL_CACHE_SIZE,
                                0,
                                socket_id);

    session_priv_pool = session_pool;

#endif

/* Configure the crypto device. */
struct rte_cryptodev_config conf = {
    .nb_queue_pairs = 1,
    .socket_id = socket_id
};

struct rte_cryptodev_qp_conf qp_conf = {
    .nb_descriptors = 2048,
    .mp_session = session_pool,
    .mp_session_private = session_priv_pool
};
```

(continues on next page)

(continued from previous page)

```
if (rte_cryptodev_configure(cdev_id, &conf) < 0)
    rte_exit(EXIT_FAILURE, "Failed to configure cryptodev %u", cdev_id);

if (rte_cryptodev_queue_pair_setup(cdev_id, 0, &qp_conf, socket_id) < 0)
    rte_exit(EXIT_FAILURE, "Failed to setup queue pair\n");

if (rte_cryptodev_start(cdev_id) < 0)
    rte_exit(EXIT_FAILURE, "Failed to start device\n");

/* Create the crypto transform. */
uint8_t cipher_key[16] = {0};
struct rte_crypto_sym_xform cipher_xform = {
    .next = NULL,
    .type = RTE_CRYPTO_SYM_XFORM_CIPHER,
    .cipher = {
        .op = RTE_CRYPTO_CIPHER_OP_ENCRYPT,
        .algo = RTE_CRYPTO_CIPHER_AES_CBC,
        .key = {
            .data = cipher_key,
            .length = AES_CBC_KEY_LENGTH
        },
        .iv = {
            .offset = IV_OFFSET,
            .length = AES_CBC_IV_LENGTH
        }
    }
};

/* Create crypto session and initialize it for the crypto device. */
struct rte_cryptodev_sym_session *session;
session = rte_cryptodev_sym_session_create(session_pool);
if (session == NULL)
    rte_exit(EXIT_FAILURE, "Session could not be created\n");

if (rte_cryptodev_sym_session_init(cdev_id, session,
                &cipher_xform, session_priv_pool) < 0)
    rte_exit(EXIT_FAILURE, "Session could not be initialized "
                "for the crypto device\n");

/* Get a burst of crypto operations. */
struct rte_crypto_op *crypto_ops[BURST_SIZE];
if (rte_crypto_op_bulk_alloc(crypto_op_pool,
                        RTE_CRYPTO_OP_TYPE_SYMMETRIC,
                        crypto_ops, BURST_SIZE) == 0)
    rte_exit(EXIT_FAILURE, "Not enough crypto operations available\n");

/* Get a burst of mbufs. */
struct rte_mbuf *mbufs[BURST_SIZE];
if (rte_pktmbuf_alloc_bulk(mbuf_pool, mbufs, BURST_SIZE) < 0)
    rte_exit(EXIT_FAILURE, "Not enough mbufs available");

/* Initialize the mbufs and append them to the crypto operations. */
unsigned int i;
for (i = 0; i < BURST_SIZE; i++) {
    if (rte_pktmbuf_append(mbufs[i], BUFFER_SIZE) == NULL)
        rte_exit(EXIT_FAILURE, "Not enough room in the mbuf\n");
    crypto_ops[i]->sym->m_src = mbufs[i];
}

/* Set up the crypto operations. */
```

(continues on next page)

(continued from previous page)

```
for (i = 0; i < BURST_SIZE; i++) {
    struct rte_crypto_op *op = crypto_ops[i];
    /* Modify bytes of the IV at the end of the crypto operation */
    uint8_t *iv_ptr = rte_crypto_op_ctod_offset(op, uint8_t *,
                                                IV_OFFSET);

    generate_random_bytes(iv_ptr, AES_CBC_IV_LENGTH);

    op->sym->cipher.data.offset = 0;
    op->sym->cipher.data.length = BUFFER_SIZE;

    /* Attach the crypto session to the operation */
    rte_crypto_op_attach_sym_session(op, session);
}

/* Enqueue the crypto operations in the crypto device. */
uint16_t num_enqueued_ops = rte_cryptodev_enqueue_burst(cdev_id, 0,
                                        crypto_ops, BURST_SIZE);

/*
 * Dequeue the crypto operations until all the operations
 * are processed in the crypto device.
 */
uint16_t num_dequeued_ops, total_num_dequeued_ops = 0;
do {
    struct rte_crypto_op *dequeued_ops[BURST_SIZE];
    num_dequeued_ops = rte_cryptodev_dequeue_burst(cdev_id, 0,
                                    dequeued_ops, BURST_SIZE);
    total_num_dequeued_ops += num_dequeued_ops;

    /* Check if operation was processed successfully */
    for (i = 0; i < num_dequeued_ops; i++) {
        if (dequeued_ops[i]->status != RTE_CRYPTO_OP_STATUS_SUCCESS)
            rte_exit(EXIT_FAILURE,
                    "Some operations were not processed correctly");
    }

    rte_mempool_put_bulk(crypto_op_pool, (void **)dequeued_ops,
                                        num_dequeued_ops);
} while (total_num_dequeued_ops < num_enqueued_ops);
```

5.17.8 Asymmetric Cryptography

The cryptodev library currently provides support for the following asymmetric Crypto operations; RSA, Modular exponentiation and inversion, Diffie-Hellman public and/or private key generation and shared secret compute, DSA Signature generation and verification.

Session and Session Management

Sessions are used in asymmetric cryptographic processing to store the immutable data defined in asymmetric cryptographic transform which is further used in the operation processing. Sessions typically stores information, such as, public and private key information or domain params or prime modulus data i.e. immutable across data sets. Crypto sessions cache this immutable data in a optimal way for the underlying PMD and this allows further acceleration of the offload of Crypto workloads.

Like symmetric, the Crypto device framework provides APIs to allocate and initialize asymmetric sessions for crypto devices, where sessions are mempool objects. It is the application's responsibility to create and manage the session mempools. Application using both symmetric and asymmetric sessions should allocate and maintain different sessions pools for each type.

An application can use `rte_cryptodev_get_asym_session_private_size()` to get the private size of asymmetric session on a given crypto device. This function would allow an application to calculate the max device asymmetric session size of all crypto devices to create a single session mempool. If instead an application creates multiple asymmetric session mempools, the Crypto device framework also provides `rte_cryptodev_asym_get_header_session_size()` to get the size of an uninitialized session.

Once the session mempools have been created, `rte_cryptodev_asym_session_create()` is used to allocate an uninitialized asymmetric session from the given mempool. The session then must be initialized using `rte_cryptodev_asym_session_init()` for each of the required crypto devices. An asymmetric transform chain is used to specify the operation and its parameters. See the section below for details on transforms.

When a session is no longer used, user must call `rte_cryptodev_asym_session_clear()` for each of the crypto devices that are using the session, to free all driver private asymmetric session data. Once this is done, session should be freed using `rte_cryptodev_asym_session_free()` which returns them to their mempool.

Asymmetric Sessionless Support

Asymmetric crypto framework supports session-less operations as well.

Fields that should be set by user are:

Member xform of struct rte_crypto_asym_op should point to the user created rte_crypto_asym_xform. Note that rte_crypto_asym_xform should be immutable for the lifetime of associated crypto_op.

Member sess_type of rte_crypto_op should also be set to RTE_CRYPTO_OP_SESSIONLESS.

Transforms and Transform Chaining

Asymmetric Crypto transforms (`rte_crypto_asym_xform`) are the mechanism used to specify the details of the asymmetric Crypto operation. Next pointer within xform allows transform to be chained together. Also it is important to note that the order in which the transforms are passed indicates the order of the chaining. Allocation of the xform structure is in the application domain. To allow future API extensions in a backwardly compatible manner, e.g. addition of a new parameter, the application should zero the full xform struct before populating it.

Not all asymmetric crypto xforms are supported for chaining. Currently supported asymmetric crypto chaining is Diffie-Hellman private key generation followed by public generation. Also, currently API does not support chaining of symmetric and asymmetric crypto xforms.

Each xform defines specific asymmetric crypto algo. Currently supported are: * RSA * Modular operations (Exponentiation and Inverse) * Diffie-Hellman * DSA * None - special case where PMD may support a passthrough mode. More for diagnostic purpose

See *DPDK API Reference* for details on each rte_crypto_xxx_xform struct

Asymmetric Operations

The asymmetric Crypto operation structure contains all the mutable data relating to asymmetric cryptographic processing on an input data buffer. It uses either RSA, Modular, Diffie-Hellman or DSA operations depending upon session it is attached to.

Every operation must carry a valid session handle which further carries information on xform or xform-chain to be performed on op. Every xform type defines its own set of operational params in their respective rte_crypto_xxx_op_param struct. Depending on xform information within session, PMD picks up and process respective op_param struct. Unlike symmetric, asymmetric operations do not use mbufs for input/output. They operate on data buffer of type `rte_crypto_param`.

See *DPDK API Reference* for details on each rte_crypto_xxx_op_param struct

5.17.9 Asymmetric crypto Sample code

There's a unit test application test_cryptodev_asym.c inside unit test framework that show how to setup and process asymmetric operations using cryptodev library.

The following sample code shows the basic steps to compute modular exponentiation using 1024-bit modulus length using openssl PMD available in DPDK (performing other crypto operations is similar except change to respective op and xform setup).

```
/*
 * Simple example to compute modular exponentiation with 1024-bit key
 *
 */
#define MAX_ASYM_SESSIONS   10
#define NUM_ASYM_BUFS       10

struct rte_mempool *crypto_op_pool, *asym_session_pool;
unsigned int asym_session_size;
int ret;

/* Initialize EAL. */
ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");

uint8_t socket_id = rte_socket_id();

/* Create crypto operation pool. */
crypto_op_pool = rte_crypto_op_pool_create(
                                "crypto_op_pool",
                                RTE_CRYPTO_OP_TYPE_ASYMMETRIC,
                                NUM_ASYM_BUFS, 0, 0,
                                socket_id);
if (crypto_op_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot create crypto op pool\n");

/* Create the virtual crypto device. */
```

(continues on next page)

(continued from previous page)

```
char args[128];
const char *crypto_name = "crypto_openssl";
snprintf(args, sizeof(args), "socket_id=%d", socket_id);
ret = rte_vdev_init(crypto_name, args);
if (ret != 0)
    rte_exit(EXIT_FAILURE, "Cannot create virtual device");

uint8_t cdev_id = rte_cryptodev_get_dev_id(crypto_name);

/* Get private asym session data size. */
asym_session_size = rte_cryptodev_get_asym_private_session_size(cdev_id);

/*
 * Create session mempool, with two objects per session,
 * one for the session header and another one for the
 * private asym session data for the crypto device.
 */
asym_session_pool = rte_mempool_create("asym_session_pool",
                                MAX_ASYM_SESSIONS * 2,
                                asym_session_size,
                                0,
                                0, NULL, NULL, NULL,
                                NULL, socket_id,
                                0);

/* Configure the crypto device. */
struct rte_cryptodev_config conf = {
    .nb_queue_pairs = 1,
    .socket_id = socket_id
};
struct rte_cryptodev_qp_conf qp_conf = {
    .nb_descriptors = 2048
};

if (rte_cryptodev_configure(cdev_id, &conf) < 0)
    rte_exit(EXIT_FAILURE, "Failed to configure cryptodev %u", cdev_id);

if (rte_cryptodev_queue_pair_setup(cdev_id, 0, &qp_conf,
                        socket_id, asym_session_pool) < 0)
    rte_exit(EXIT_FAILURE, "Failed to setup queue pair\n");

if (rte_cryptodev_start(cdev_id) < 0)
    rte_exit(EXIT_FAILURE, "Failed to start device\n");

/* Setup crypto xform to do modular exponentiation with 1024 bit
     * length modulus
     */
struct rte_crypto_asym_xform modex_xform = {
            .next = NULL,
            .xform_type = RTE_CRYPTO_ASYM_XFORM_MODEX,
            .modex = {
                    .modulus = {
                            .data =
                            (uint8_t *)
                            ("\xb3\xa1\xaf\xb7\x13\x08\x00\x0a\x35\xdc\x2b\x20\x8d"
                            "\xa1\xb5\xce\x47\x8a\xc3\x80\xf4\x7d\x4a\xa2\x62\xfd\x61\x7f"
                            "\xb5\xa8\xde\x0a\x17\x97\xa0\xbf\xdf\x56\x5a\x3d\x51\x56\x4f"
                            "\x70\x70\x3f\x63\x6a\x44\x5b\xad\x84\x0d\x3f\x27\x6e\x3b\x34"
                            "\x91\x60\x14\xb9\xaa\x72\xfd\xa3\x64\xd2\x03\xa7\x53\x87\x9e"
                            "\x88\x0b\xc1\x14\x93\x1a\x62\xff\xb1\x5d\x74\xcd\x59\x63\x18"
                            "\x11\x3d\x4f\xba\x75\xd4\x33\x4e\x23\x6b\x7b\x57\x44\xe1\xd3"
```

(continues on next page)

(continued from previous page)

```
                        "\x03\x13\xa6\xf0\x8b\x60\xb0\x9e\xee\x75\x08\x9d\x71\x63\x13"
                        "\xcb\xa6\x81\x92\x14\x03\x22\x2d\xde\x55"),
                        .length = 128
                },
                .exponent = {
                        .data = (uint8_t *)("\x01\x00\x01"),
                        .length = 3
                }
        }
};
/* Create asym crypto session and initialize it for the crypto device. */
struct rte_cryptodev_asym_session *asym_session;
asym_session = rte_cryptodev_asym_session_create(asym_session_pool);
if (asym_session == NULL)
    rte_exit(EXIT_FAILURE, "Session could not be created\n");

if (rte_cryptodev_asym_session_init(cdev_id, asym_session,
                &modex_xform, asym_session_pool) < 0)
    rte_exit(EXIT_FAILURE, "Session could not be initialized "
                "for the crypto device\n");

/* Get a burst of crypto operations. */
struct rte_crypto_op *crypto_ops[1];
if (rte_crypto_op_bulk_alloc(crypto_op_pool,
                        RTE_CRYPTO_OP_TYPE_ASYMMETRIC,
                        crypto_ops, 1) == 0)
    rte_exit(EXIT_FAILURE, "Not enough crypto operations available\n");

/* Set up the crypto operations. */
struct rte_crypto_asym_op *asym_op = crypto_ops[0]->asym;

    /* calculate mod exp of value 0xf8 */
static unsigned char base[] = {0xF8};
asym_op->modex.base.data = base;
asym_op->modex.base.length = sizeof(base);
    asym_op->modex.base.iova = base;

/* Attach the asym crypto session to the operation */
rte_crypto_op_attach_asym_session(op, asym_session);

/* Enqueue the crypto operations in the crypto device. */
uint16_t num_enqueued_ops = rte_cryptodev_enqueue_burst(cdev_id, 0,
                                        crypto_ops, 1);

/*
 * Dequeue the crypto operations until all the operations
 * are processed in the crypto device.
 */
uint16_t num_dequeued_ops, total_num_dequeued_ops = 0;
do {
    struct rte_crypto_op *dequeued_ops[1];
    num_dequeued_ops = rte_cryptodev_dequeue_burst(cdev_id, 0,
                                    dequeued_ops, 1);
    total_num_dequeued_ops += num_dequeued_ops;

    /* Check if operation was processed successfully */
    if (dequeued_ops[0]->status != RTE_CRYPTO_OP_STATUS_SUCCESS)
            rte_exit(EXIT_FAILURE,
                    "Some operations were not processed correctly");

} while (total_num_dequeued_ops < num_enqueued_ops);
```

Asymmetric Crypto Device API

The cryptodev Library API is described in the DPDK API Reference

5.18 Compression Device Library

The compression framework provides a generic set of APIs to perform compression services as well as to query and configure compression devices both physical(hardware) and virtual(software) to perform those services. The framework currently only supports lossless compression schemes: Deflate and LZS.

5.18.1 Device Management

Device Creation

Physical compression devices are discovered during the bus probe of the EAL function which is executed at DPDK initialization, based on their unique device identifier. For e.g. PCI devices can be identified using PCI BDF (bus/bridge, device, function). Specific physical compression devices, like other physical devices in DPDK can be white-listed or black-listed using the EAL command line options.

Virtual devices can be created by two mechanisms, either using the EAL command line options or from within the application using an EAL API directly.

From the command line using the –vdev EAL option

```
--vdev  '<pmd name>,socket_id=0'
```

Note:

- If DPDK application requires multiple software compression PMD devices then required number of `--vdev` with appropriate libraries are to be added.
- An Application with multiple compression device instances exposed by the same PMD must specify a unique name for each device.

Example: `--vdev 'pmd0' --vdev 'pmd1'`

Or, by using the rte_vdev_init API within the application code.

```
rte_vdev_init("<pmd_name>","socket_id=0")
```

All virtual compression devices support the following initialization parameters:

- `socket_id` - socket on which to allocate the device resources on.

Device Identification

Each device, whether virtual or physical is uniquely designated by two identifiers:

- A unique device index used to designate the compression device in all functions exported by the compressdev API.
- A device name used to designate the compression device in console messages, for administration or debugging purposes.

Device Configuration

The configuration of each compression device includes the following operations:

- Allocation of resources, including hardware resources if a physical device.
- Resetting the device into a well-known default state.
- Initialization of statistics counters.

The `rte_compressdev_configure` API is used to configure a compression device.

The `rte_compressdev_config` structure is used to pass the configuration parameters.

See *DPDK API Reference* for details.

Configuration of Queue Pairs

Each compression device queue pair is individually configured through the `rte_compressdev_queue_pair_setup` API.

The `max_inflight_ops` is used to pass maximum number of rte_comp_op that could be present in a queue at-a-time. PMD then can allocate resources accordingly on a specified socket.

See *DPDK API Reference* for details.

Logical Cores, Memory and Queues Pair Relationships

Library supports NUMA similarly as described in Cryptodev library section.

A queue pair cannot be shared and should be exclusively used by a single processing context for enqueuing operations or dequeuing operations on the same compression device since sharing would require global locks and hinder performance. It is however possible to use a different logical core to dequeue an operation on a queue pair from the logical core on which it was enqueued. This means that a compression burst enqueue/dequeue APIs are a logical place to transition from one logical core to another in a data processing pipeline.

5.18.2 Device Features and Capabilities

Compression devices define their functionality through two mechanisms, global device features and algorithm features. Global devices features identify device wide level features which are applicable to the whole device such as supported hardware acceleration and CPU features. List of compression device features can be seen in the RTE_COMPDEV_FF_XXX macros.

The algorithm features lists individual algo feature which device supports per-algorithm, such as a stateful compression/decompression, checksums operation etc. List of algorithm features can be seen in the RTE_COMP_FF_XXX macros.

Capabilities

Each PMD has a list of capabilities, including algorithms listed in enum `rte_comp_algorithm` and its associated feature flag and sliding window range in log base 2 value. Sliding window tells the minimum and maximum size of lookup window that algorithm uses to find duplicates.

See *DPDK API Reference* for details.

Each Compression poll mode driver defines its array of capabilities for each algorithm it supports. See PMD implementation for capability initialization.

Capabilities Discovery

PMD capability and features are discovered via `rte_compressdev_info_get` function.

The `rte_compressdev_info` structure contains all the relevant information for the device.

See *DPDK API Reference* for details.

5.18.3 Compression Operation

DPDK compression supports two types of compression methodologies:

- Stateless, data associated to a compression operation is compressed without any reference to another compression operation.
- Stateful, data in each compression operation is compressed with reference to previous compression operations in the same data stream i.e. history of data is maintained between the operations.

For more explanation, please refer RFC https://www.ietf.org/rfc/rfc1951.txt

Operation Representation

Compression operation is described via `struct rte_comp_op`, which contains both input and output data. The operation structure includes the operation type (stateless or stateful), the operation status and the priv_xform/stream handle, source, destination and checksum buffer pointers. It also contains the source mempool from which the operation is allocated. PMD updates consumed field with amount of data read from source buffer and produced field with amount of data of written into destination buffer along with status of operation. See section *Produced, Consumed And Operation Status* for more details.

Compression operations mempool also has an ability to allocate private memory with the operation for application's purposes. Application software is responsible for specifying all the operation specific fields

in the `rte_comp_op` structure which are then used by the compression PMD to process the requested operation.

Operation Management and Allocation

The compressdev library provides an API set for managing compression operations which utilize the Mempool Library to allocate operation buffers. Therefore, it ensures that the compression operation is interleaved optimally across the channels and ranks for optimal processing.

A `rte_comp_op` contains a field indicating the pool it originated from.

`rte_comp_op_alloc()` and `rte_comp_op_bulk_alloc()` are used to allocate compression operations from a given compression operation mempool. The operation gets reset before being returned to a user so that operation is always in a good known state before use by the application.

`rte_comp_op_free()` is called by the application to return an operation to its allocating pool.

See *DPDK API Reference* for details.

Passing source data as mbuf-chain

If input data is scattered across several different buffers, then Application can either parse through all such buffers and make one mbuf-chain and enqueue it for processing or, alternatively, it can make multiple sequential enqueue_burst() calls for each of them processing them statefully. See *Compression API Stateful Operation* for stateful processing of ops.

Operation Status

Each operation carries a status information updated by PMD after it is processed. Following are currently supported:

- **RTE_COMP_OP_STATUS_SUCCESS,**
  Operation is successfully completed
- **RTE_COMP_OP_STATUS_NOT_PROCESSED,**
  Operation has not yet been processed by the device
- **RTE_COMP_OP_STATUS_INVALID_ARGS,**
  Operation failed due to invalid arguments in request
- **RTE_COMP_OP_STATUS_ERROR,**
  Operation failed because of internal error
- **RTE_COMP_OP_STATUS_INVALID_STATE,**
  Operation is invoked in invalid state
- **RTE_COMP_OP_STATUS_OUT_OF_SPACE_TERMINATED,**
  Output buffer ran out of space during processing. Error case, PMD cannot continue from here.
- **RTE_COMP_OP_STATUS_OUT_OF_SPACE_RECOVERABLE,**
  Output buffer ran out of space before operation completed, but this is not an error case. Output data up to op.produced can be used and next op in the stream should continue on from op.consumed+1.

Operation status after enqueue / dequeue

Some of the above values may arise in the op after an `rte_compressdev_enqueue_burst()`. If number ops enqueued < number ops requested then the app should check the op.status of nb_enqd+1. If status is RTE_COMP_OP_STATUS_NOT_PROCESSED, it likely indicates a full-queue case for a hardware device and a retry after dequeuing some ops is likely to be successful. If the op holds any other status, e.g. RTE_COMP_OP_STATUS_INVALID_ARGS, a retry with the same op is unlikely to be successful.

Produced, Consumed And Operation Status

- **If status is RTE_COMP_OP_STATUS_SUCCESS,**
  consumed = amount of data read from input buffer, and produced = amount of data written in destination buffer
- **If status is RTE_COMP_OP_STATUS_ERROR,**
  consumed = produced = undefined
- **If status is RTE_COMP_OP_STATUS_OUT_OF_SPACE_TERMINATED,**
  consumed = 0 and produced = usually 0, but in decompression cases a PMD may return > 0 i.e. amount of data successfully produced until out of space condition hit. Application can consume output data in this case, if required.
- **If status is RTE_COMP_OP_STATUS_OUT_OF_SPACE_RECOVERABLE,**
  consumed = amount of data read, and produced = amount of data successfully produced until out of space condition hit. PMD has ability to recover from here, so application can submit next op from consumed+1 and a destination buffer with available space.

5.18.4 Transforms

Compression transforms (`rte_comp_xform`) are the mechanism to specify the details of the compression operation such as algorithm, window size and checksum.

5.18.5 Compression API Hash support

Compression API allows application to enable digest calculation alongside compression and decompression of data. A PMD reflects its support for hash algorithms via capability algo feature flags. If supported, PMD calculates digest always on plaintext i.e. before compression and after decompression.

Currently supported list of hash algos are SHA-1 and SHA2 family SHA256.

See *DPDK API Reference* for details.

If required, application should set valid hash algo in compress or decompress xforms during `rte_compressdev_stream_create()` or `rte_compressdev_private_xform_create()` and pass a valid output buffer in `rte_comp_op` hash field struct to store the resulting digest. Buffer passed should be contiguous and large enough to store digest which is 20 bytes for SHA-1 and 32 bytes for SHA2-256.

5.18.6 Compression API Stateless operation

An op is processed stateless if it has - op_type set to RTE_COMP_OP_STATELESS - flush value set to RTE_COMP_FLUSH_FULL or RTE_COMP_FLUSH_FINAL (required only on compression side), - All required input in source buffer

When all of the above conditions are met, PMD initiates stateless processing and releases acquired resources after processing of current operation is complete. Application can enqueue multiple stateless ops in a single burst and must attach priv_xform handle to such ops.

priv_xform in Stateless operation

priv_xform is PMD internally managed private data that it maintains to do stateless processing. priv_xforms are initialized provided a generic xform structure by an application via making call to `rte_compressdev_private_xform_create`, at an output PMD returns an opaque priv_xform reference. If PMD support SHAREABLE priv_xform indicated via algorithm feature flag, then application can attach same priv_xform with many stateless ops at-a-time. If not, then application needs to create as many priv_xforms as it expects to have stateless operations in-flight.

Fig. 5.26: Stateless Ops using Non-Shareable priv_xform

Fig. 5.27: Stateless Ops using Shareable priv_xform

Application should call `rte_compressdev_private_xform_create()` and attach to stateless op before enqueuing them for processing and free via `rte_compressdev_private_xform_free()` during termination.

An example pseudocode to setup and process NUM_OPS stateless ops with each of length OP_LEN using priv_xform would look like:

```
/*
 * pseudocode for stateless compression
 */

uint8_t cdev_id = rte_compressdev_get_dev_id(<pmd name>);

/* configure the device. */
if (rte_compressdev_configure(cdev_id, &conf) < 0)
    rte_exit(EXIT_FAILURE, "Failed to configure compressdev %u", cdev_id);

if (rte_compressdev_queue_pair_setup(cdev_id, 0, NUM_MAX_INFLIGHT_OPS,
                        socket_id()) < 0)
    rte_exit(EXIT_FAILURE, "Failed to setup queue pair\n");

if (rte_compressdev_start(cdev_id) < 0)
    rte_exit(EXIT_FAILURE, "Failed to start device\n");

/* setup compress transform */
struct rte_comp_xform compress_xform = {
    .type = RTE_COMP_COMPRESS,
    .compress = {
        .algo = RTE_COMP_ALGO_DEFLATE,
        .deflate = {
            .huffman = RTE_COMP_HUFFMAN_DEFAULT
```

(continues on next page)

(continued from previous page)

```
        },
        .level = RTE_COMP_LEVEL_PMD_DEFAULT,
        .chksum = RTE_COMP_CHECKSUM_NONE,
        .window_size = DEFAULT_WINDOW_SIZE,
        .hash_algo = RTE_COMP_HASH_ALGO_NONE
    }
};

/* create priv_xform and initialize it for the compression device. */
rte_compressdev_info dev_info;
void *priv_xform = NULL;
int shareable = 1;
rte_compressdev_info_get(cdev_id, &dev_info);
if (dev_info.capabilities->comp_feature_flags & RTE_COMP_FF_SHAREABLE_PRIV_XFORM) {
    rte_compressdev_private_xform_create(cdev_id, &compress_xform, &priv_xform);
} else {
    shareable = 0;
}

/* create operation pool via call to rte_comp_op_pool_create and alloc ops */
struct rte_comp_op *comp_ops[NUM_OPS];
rte_comp_op_bulk_alloc(op_pool, comp_ops, NUM_OPS);

/* prepare ops for compression operations */
for (i = 0; i < NUM_OPS; i++) {
    struct rte_comp_op *op = comp_ops[i];
    if (!shareable)
        rte_compressdev_private_xform_create(cdev_id, &compress_xform, &op->priv_xform)
    else
        op->private_xform = priv_xform;
    op->op_type = RTE_COMP_OP_STATELESS;
    op->flush_flag = RTE_COMP_FLUSH_FINAL;

    op->src.offset = 0;
    op->dst.offset = 0;
    op->src.length = OP_LEN;
    op->input_chksum = 0;
    setup op->m_src and op->m_dst;
}
num_enqd = rte_compressdev_enqueue_burst(cdev_id, 0, comp_ops, NUM_OPS);
/* wait for this to complete before enqueuing next*/
do {
    num_deque = rte_compressdev_dequeue_burst(cdev_id, 0 , &processed_ops, NUM_OPS);
} while (num_dqud < num_enqd);
```

Stateless and OUT_OF_SPACE

OUT_OF_SPACE is a condition when output buffer runs out of space and where PMD still has more data to produce. If PMD runs into such condition, then PMD returns RTE_COMP_OP_OUT_OF_SPACE_TERMINATED error. In such case, PMD resets itself and can set consumed=0 and produced=amount of output it could produce before hitting out_of_space. Application would need to resubmit the whole input with a larger output buffer, if it wants the operation to be completed.

Hash in Stateless

If hash is enabled, digest buffer will contain valid data after op is successfully processed i.e. dequeued with status = RTE_COMP_OP_STATUS_SUCCESS.

Checksum in Stateless

If checksum is enabled, checksum will only be available after op is successfully processed i.e. dequeued with status = RTE_COMP_OP_STATUS_SUCCESS.

5.18.7 Compression API Stateful operation

Compression API provide RTE_COMP_FF_STATEFUL_COMPRESSION and RTE_COMP_FF_STATEFUL_DECOMPRESSION feature flag for PMD to reflect its support for Stateful operations.

A Stateful operation in DPDK compression means application invokes enqueue burst() multiple times to process related chunk of data because application broke data into several ops.

In such case - ops are setup with op_type RTE_COMP_OP_STATEFUL, - all ops except last set to flush value = RTE_COMP_FLUSH_NONE/SYNC and last set to flush value RTE_COMP_FLUSH_FULL/FINAL.

In case of either one or all of the above conditions, PMD initiates stateful processing and releases acquired resources after processing operation with flush value = RTE_COMP_FLUSH_FULL/FINAL is complete. Unlike stateless, application can enqueue only one stateful op from a particular stream at a time and must attach stream handle to each op.

Stream in Stateful operation

stream in DPDK compression is a logical entity which identifies related set of ops, say, a one large file broken into multiple chunks then file is represented by a stream and each chunk of that file is represented by compression op *rte_comp_op*. Whenever application wants a stateful processing of such data, then it must get a stream handle via making call to `rte_compressdev_stream_create()` with xform, at an output the target PMD will return an opaque stream handle to application which it must attach to all of the ops carrying data of that stream. In stateful processing, every op requires previous op data for compression/decompression. A PMD allocates and set up resources such as history, states, etc. within a stream, which are maintained during the processing of the related ops.

Unlike priv_xforms, stream is always a NON_SHAREABLE entity. One stream handle must be attached to only one set of related ops and cannot be reused until all of them are processed with status Success or failure.

Fig. 5.28: Stateful Ops

Application should call `rte_compressdev_stream_create()` and attach to op before enqueuing them for processing and free via `rte_compressdev_stream_free()` during termination. All ops that are to be processed statefully should carry *same* stream.

See *DPDK API Reference* document for details.

An example pseudocode to set up and process a stream having NUM_CHUNKS with each chunk size of CHUNK_LEN would look like:

```
/*
 * pseudocode for stateful compression
 */

uint8_t cdev_id = rte_compressdev_get_dev_id(<pmd name>);

/* configure the  device. */
if (rte_compressdev_configure(cdev_id, &conf) < 0)
    rte_exit(EXIT_FAILURE, "Failed to configure compressdev %u", cdev_id);

if (rte_compressdev_queue_pair_setup(cdev_id, 0, NUM_MAX_INFLIGHT_OPS,
                                socket_id()) < 0)
    rte_exit(EXIT_FAILURE, "Failed to setup queue pair\n");

if (rte_compressdev_start(cdev_id) < 0)
    rte_exit(EXIT_FAILURE, "Failed to start device\n");

/* setup compress transform. */
struct rte_comp_xform compress_xform = {
    .type = RTE_COMP_COMPRESS,
    .compress = {
        .algo = RTE_COMP_ALGO_DEFLATE,
        .deflate = {
            .huffman = RTE_COMP_HUFFMAN_DEFAULT
        },
        .level = RTE_COMP_LEVEL_PMD_DEFAULT,
        .chksum = RTE_COMP_CHECKSUM_NONE,
        .window_size = DEFAULT_WINDOW_SIZE,
        .hash_algo = RTE_COMP_HASH_ALGO_NONE
    }
};

/* create stream */
void *stream;
rte_compressdev_stream_create(cdev_id, &compress_xform, &stream);

/* create an op pool and allocate ops */
rte_comp_op_bulk_alloc(op_pool, comp_ops, NUM_CHUNKS);

/* Prepare source and destination mbufs for compression operations */
unsigned int i;
for (i = 0; i < NUM_CHUNKS; i++) {
    if (rte_pktmbuf_append(mbufs[i], CHUNK_LEN) == NULL)
        rte_exit(EXIT_FAILURE, "Not enough room in the mbuf\n");
    comp_ops[i]->m_src = mbufs[i];
    if (rte_pktmbuf_append(dst_mbufs[i], CHUNK_LEN) == NULL)
        rte_exit(EXIT_FAILURE, "Not enough room in the mbuf\n");
    comp_ops[i]->m_dst = dst_mbufs[i];
}

/* Set up the compress operations. */
for (i = 0; i < NUM_CHUNKS; i++) {
    struct rte_comp_op *op = comp_ops[i];
    op->stream = stream;
    op->m_src = src_buf[i];
    op->m_dst = dst_buf[i];
    op->op_type = RTE_COMP_OP_STATEFUL;
    if (i == NUM_CHUNKS-1) {
        /* set to final, if last chunk*/
```

(continues on next page)

(continued from previous page)

```
        op->flush_flag = RTE_COMP_FLUSH_FINAL;
    } else {
        /* set to NONE, for all intermediary ops */
        op->flush_flag = RTE_COMP_FLUSH_NONE;
    }
    op->src.offset = 0;
    op->dst.offset = 0;
    op->src.length = CHUNK_LEN;
    op->input_chksum = 0;
    num_enqd = rte_compressdev_enqueue_burst(cdev_id, 0, &op[i], 1);
    /* wait for this to complete before enqueuing next*/
    do {
        num_deqd = rte_compressdev_dequeue_burst(cdev_id, 0 , &processed_ops, 1);
    } while (num_deqd < num_enqd);
    /* analyze the amount of consumed and produced data before pushing next op*/
}
```

Stateful and OUT_OF_SPACE

If PMD supports stateful operation, then OUT_OF_SPACE status is not an actual error for the PMD. In such case, PMD returns with status RTE_COMP_OP_STATUS_OUT_OF_SPACE_RECOVERABLE with consumed = number of input bytes read and produced = length of complete output buffer. Application should enqueue next op with source starting at consumed+1 and an output buffer with available space.

Hash in Stateful

If enabled, digest buffer will contain valid digest after last op in stream (having flush = RTE_COMP_FLUSH_FINAL) is successfully processed i.e. dequeued with status = RTE_COMP_OP_STATUS_SUCCESS.

Checksum in Stateful

If enabled, checksum will only be available after last op in stream (having flush = RTE_COMP_FLUSH_FINAL) is successfully processed i.e. dequeued with status = RTE_COMP_OP_STATUS_SUCCESS.

5.18.8 Burst in compression API

Scheduling of compression operations on DPDK's application data path is performed using a burst oriented asynchronous API set. A queue pair on a compression device accepts a burst of compression operations using enqueue burst API. On physical devices the enqueue burst API will place the operations to be processed on the device's hardware input queue, for virtual devices the processing of the operations is usually completed during the enqueue call to the compression device. The dequeue burst API will retrieve any processed operations available from the queue pair on the compression device, from physical devices this is usually directly from the devices processed queue, and for virtual device's from a `rte_ring` where processed operations are placed after being processed on the enqueue call.

A burst in DPDK compression can be a combination of stateless and stateful operations with a condition that for stateful ops only one op at-a-time should be enqueued from a particular stream i.e. no-two ops

should belong to same stream in a single burst. However a burst may contain multiple stateful ops as long as each op is attached to a different stream i.e. a burst can look like:

enqueue_burst	op1.no_flush	op2.no_flush	op3.flush_final	op4.no_flush	op5.no_flush

Where, op1 .. op5 all belong to different independent data units. op1, op2, op4, op5 must be stateful as stateless ops can only use flush full or final and op3 can be of type stateless or stateful. Every op with type set to RTE_COMP_OP_STATELESS must be attached to priv_xform and Every op with type set to RTE_COMP_OP_STATEFUL *must* be attached to stream.

Since each operation in a burst is independent and thus can be completed out-of-order, applications which need ordering, should setup per-op user data area with reordering information so that it can determine enqueue order at dequeue.

Also if multiple threads calls enqueue_burst() on same queue pair then it's application onus to use proper locking mechanism to ensure exclusive enqueuing of operations.

Enqueue / Dequeue Burst APIs

The burst enqueue API uses a compression device identifier and a queue pair identifier to specify the compression device queue pair to schedule the processing on. The `nb_ops` parameter is the number of operations to process which are supplied in the `ops` array of `rte_comp_op` structures. The enqueue function returns the number of operations it actually enqueued for processing, a return value equal to `nb_ops` means that all packets have been enqueued.

The dequeue API uses the same format as the enqueue API but the `nb_ops` and `ops` parameters are now used to specify the max processed operations the user wishes to retrieve and the location in which to store them. The API call returns the actual number of processed operations returned, this can never be larger than `nb_ops`.

5.18.9 Sample code

There are unit test applications that show how to use the compressdev library inside app/test/test_compressdev.c

Compression Device API

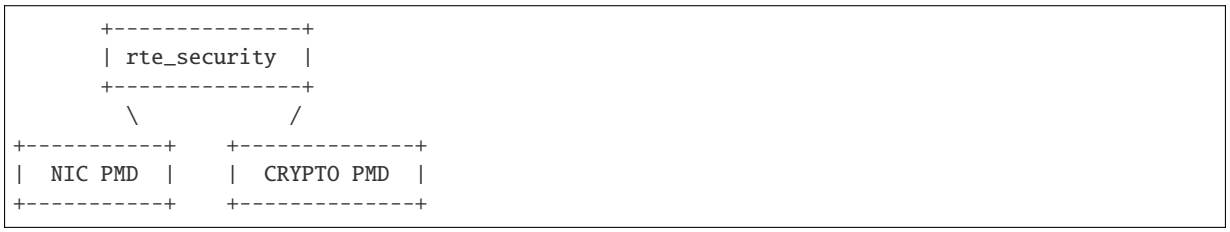
The compressdev Library API is described in the *DPDK API Reference* document.

5.19 Security Library

The security library provides a framework for management and provisioning of security protocol operations offloaded to hardware based devices. The library defines generic APIs to create and free security sessions which can support full protocol offload as well as inline crypto operation with NIC or crypto devices. The framework currently only supports the IPsec and PDCP protocol and associated operations, other protocols will be added in future.

5.19.1 Design Principles

The security library provides an additional offload capability to an existing crypto device and/or ethernet device.



Note: Currently, the security library does not support the case of multi-process. It will be updated in the future releases.

The supported offload types are explained in the sections below.

Inline Crypto

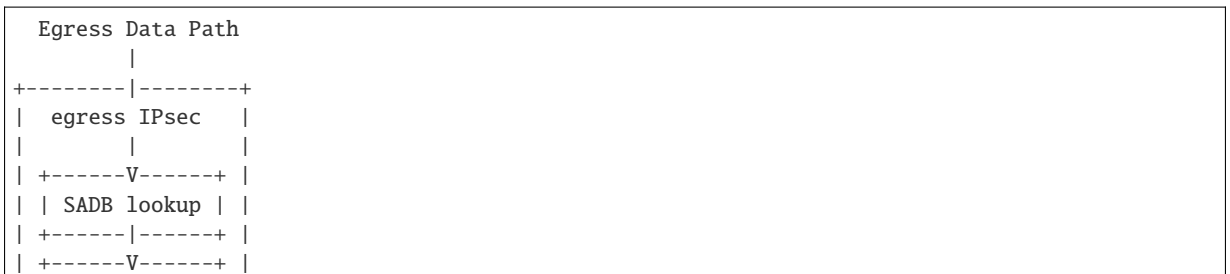
RTE_SECURITY_ACTION_TYPE_INLINE_CRYPTO: The crypto processing for security protocol (e.g. IPsec) is processed inline during receive and transmission on NIC port. The flow based security action should be configured on the port.

Ingress Data path - The packet is decrypted in RX path and relevant crypto status is set in Rx descriptors. After the successful inline crypto processing the packet is presented to host as a regular Rx packet however all security protocol related headers are still attached to the packet. e.g. In case of IPsec, the IPsec tunnel headers (if any), ESP/AH headers will remain in the packet but the received packet contains the decrypted data where the encrypted data was when the packet arrived. The driver Rx path check the descriptors and based on the crypto status sets additional flags in the rte_mbuf.ol_flags field.

Note: The underlying device may not support crypto processing for all ingress packet matching to a particular flow (e.g. fragmented packets), such packets will be passed as encrypted packets. It is the responsibility of application to process such encrypted packets using other crypto driver instance.

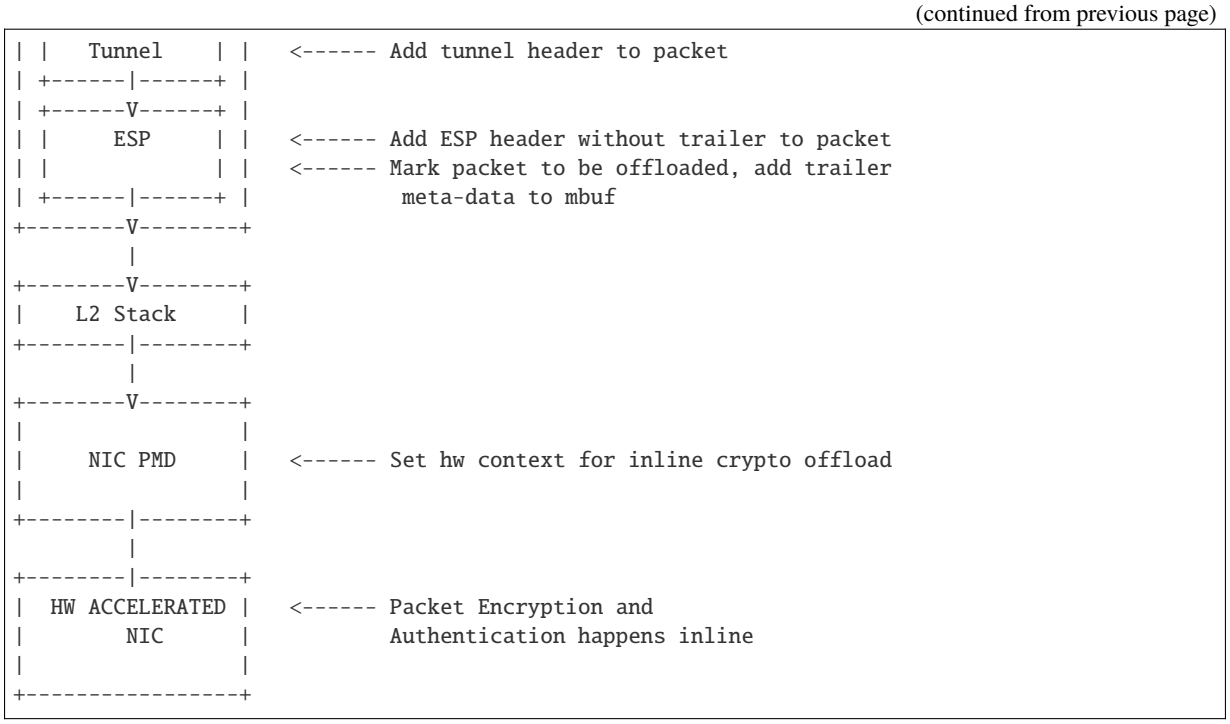
Egress Data path - The software prepares the egress packet by adding relevant security protocol headers. Only the data will not be encrypted by the software. The driver will accordingly configure the tx descriptors. The hardware device will encrypt the data before sending the packet out.

Note: The underlying device may support post encryption TSO.



(continues on next page)

(continued from previous page)



Inline protocol offload

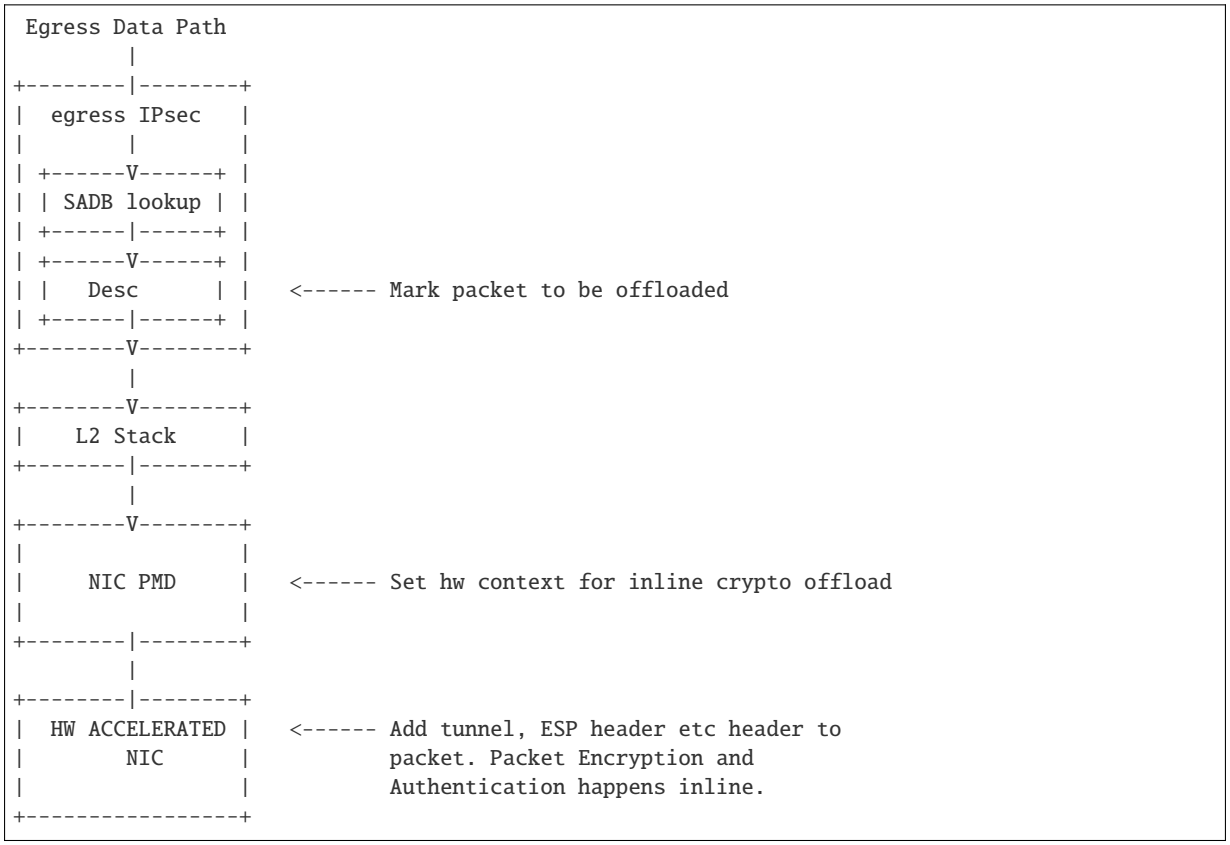
RTE_SECURITY_ACTION_TYPE_INLINE_PROTOCOL: The crypto and protocol processing for security protocol (e.g. IPsec) is processed inline during receive and transmission. The flow based security action should be configured on the port.

Ingress Data path - The packet is decrypted in the RX path and relevant crypto status is set in the Rx descriptors. After the successful inline crypto processing the packet is presented to the host as a regular Rx packet but all security protocol related headers are optionally removed from the packet. e.g. in the case of IPsec, the IPsec tunnel headers (if any), ESP/AH headers will be removed from the packet and the received packet will contains the decrypted packet only. The driver Rx path checks the descriptors and based on the crypto status sets additional flags in `rte_mbuf.ol_flags` field. The driver would also set device-specific metadata in `rte_mbuf.udata64` field. This will allow the application to identify the security processing done on the packet.

Note: The underlying device in this case is stateful. It is expected that the device shall support crypto processing for all kind of packets matching to a given flow, this includes fragmented packets (post reassembly). E.g. in case of IPsec the device may internally manage anti-replay etc. It will provide a configuration option for anti-replay behavior i.e. to drop the packets or pass them to driver with error flags set in the descriptor.

Egress Data path - The software will send the plain packet without any security protocol headers added to the packet. The driver will configure the security index and other requirement in tx descriptors. The hardware device will do security processing on the packet that includes adding the relevant protocol headers and encrypting the data before sending the packet out. The software should make sure that the buffer has required head room and tail room for any protocol header addition. The software may also do early fragmentation if the resultant packet is expected to cross the MTU size.

Note: The underlying device will manage state information required for egress processing. E.g. in case of IPsec, the seq number will be added to the packet, however the device shall provide indication when the sequence number is about to overflow. The underlying device may support post encryption TSO.



Lookaside protocol offload

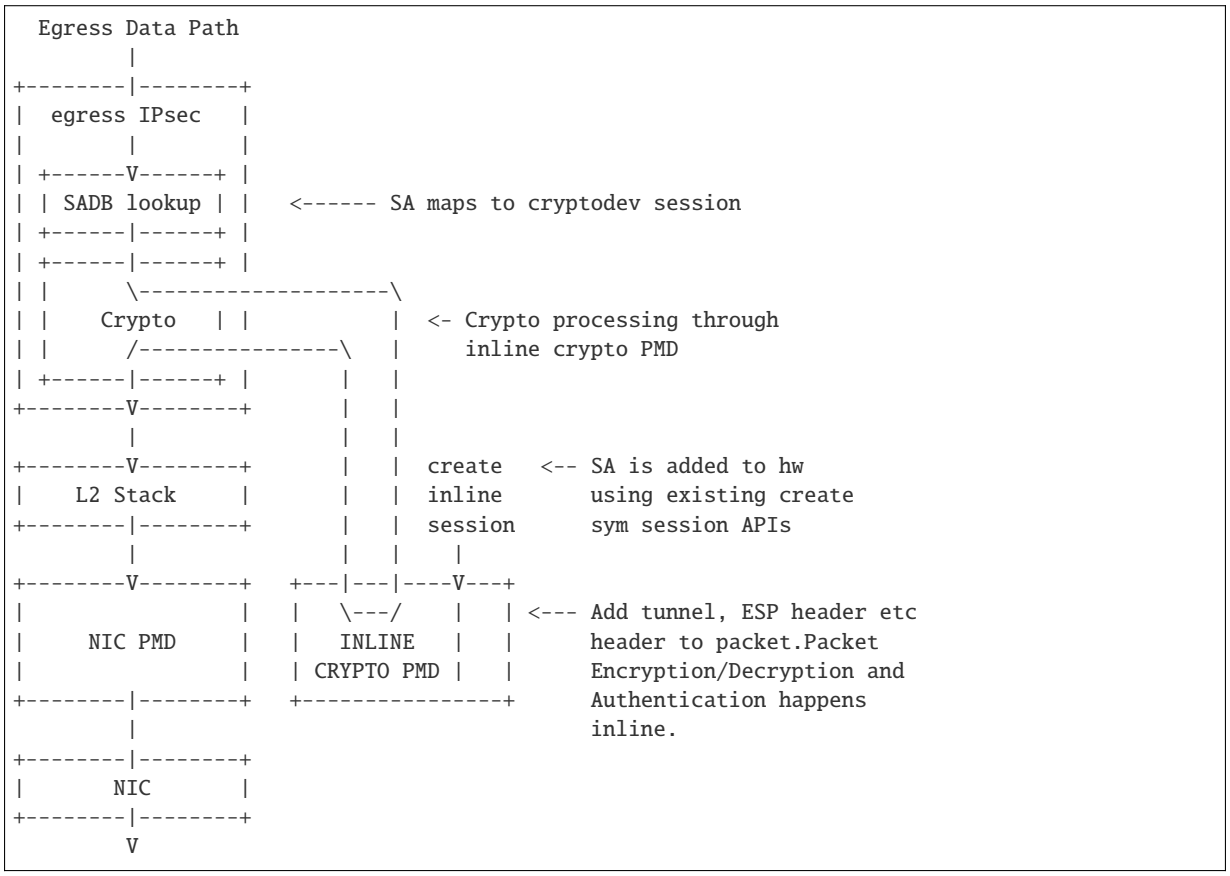
RTE_SECURITY_ACTION_TYPE_LOOKASIDE_PROTOCOL: This extends librte_cryptodev to support the programming of IPsec Security Association (SA) as part of a crypto session creation including the definition. In addition to standard crypto processing, as defined by the cryptodev, the security protocol processing is also offloaded to the crypto device.

Decryption: The packet is sent to the crypto device for security protocol processing. The device will decrypt the packet and it will also optionally remove additional security headers from the packet. E.g. in case of IPsec, IPsec tunnel headers (if any), ESP/AH headers will be removed from the packet and the decrypted packet may contain plain data only.

Note: In case of IPsec the device may internally manage anti-replay etc. It will provide a configuration option for anti-replay behavior i.e. to drop the packets or pass them to driver with error flags set in descriptor.

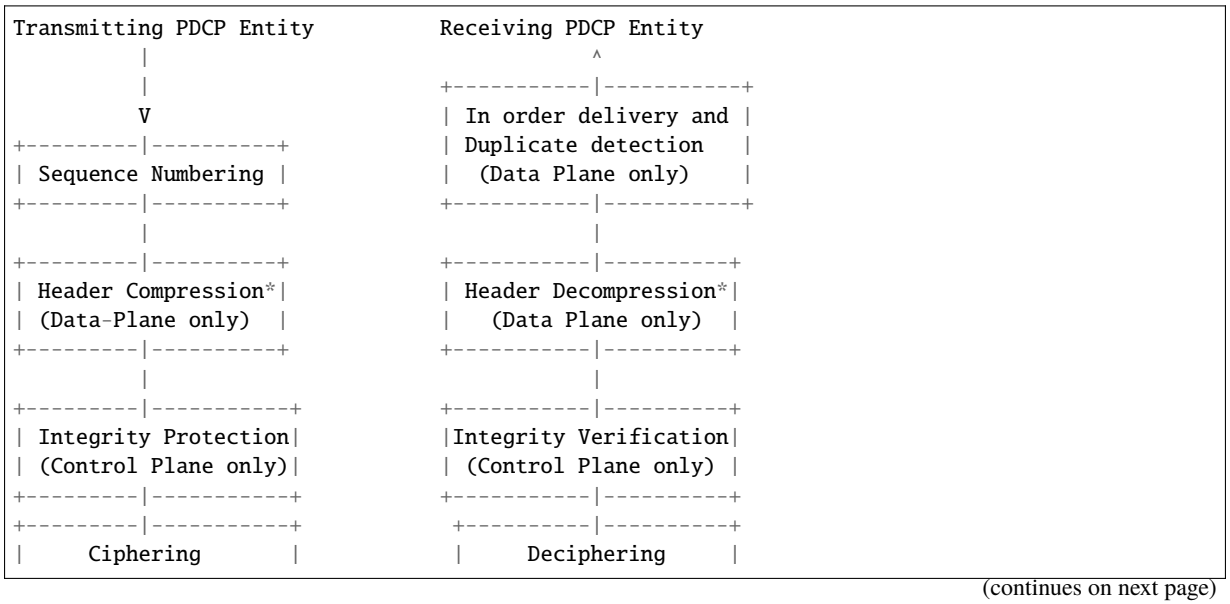
Encryption: The software will submit the packet to cryptodev as usual for encryption, the hardware device in this case will also add the relevant security protocol header along with encrypting the packet. The software should make sure that the buffer has required head room and tail room for any protocol header addition.

Note: In the case of IPsec, the seq number will be added to the packet, It shall provide an indication when the sequence number is about to overflow.



PDCP Flow Diagram

Based on 3GPP TS 36.323 Evolved Universal Terrestrial Radio Access (E-UTRA); Packet Data Convergence Protocol (PDCP) specification



(continues on next page)

(continued from previous page)

```
+---------|-----------+            +----------|----------+
+---------|-----------+            +----------|----------+
|   Add PDCP header   |            | Remove PDCP Header  |
+---------|-----------+            +----------|----------+
          |                                   |
          +------------------>>---------------+
```

Note:

- Header Compression and decompression are not supported currently.

Just like IPsec, in case of PDCP also header addition/deletion, cipher/ de-cipher, integrity protection/verification is done based on the action type chosen.

5.19.2 Device Features and Capabilities

Device Capabilities For Security Operations

The device (crypto or ethernet) capabilities which support security operations, are defined by the security action type, security protocol, protocol capabilities and corresponding crypto capabilities for security. For the full scope of the Security capability see definition of rte_security_capability structure in the *DPDK API Reference*.

```
struct rte_security_capability;
```

Each driver (crypto or ethernet) defines its own private array of capabilities for the operations it supports. Below is an example of the capabilities for a PMD which supports the IPsec and PDCP protocol.

```
static const struct rte_security_capability pmd_security_capabilities[] = {
    { /* IPsec Lookaside Protocol offload ESP Tunnel Egress */
            .action = RTE_SECURITY_ACTION_TYPE_LOOKASIDE_PROTOCOL,
            .protocol = RTE_SECURITY_PROTOCOL_IPSEC,
            .ipsec = {
                    .proto = RTE_SECURITY_IPSEC_SA_PROTO_ESP,
                    .mode = RTE_SECURITY_IPSEC_SA_MODE_TUNNEL,
                    .direction = RTE_SECURITY_IPSEC_SA_DIR_EGRESS,
                    .options = { 0 }
            },
            .crypto_capabilities = pmd_capabilities
    },
    { /* IPsec Lookaside Protocol offload ESP Tunnel Ingress */
            .action = RTE_SECURITY_ACTION_TYPE_LOOKASIDE_PROTOCOL,
            .protocol = RTE_SECURITY_PROTOCOL_IPSEC,
            .ipsec = {
                    .proto = RTE_SECURITY_IPSEC_SA_PROTO_ESP,
                    .mode = RTE_SECURITY_IPSEC_SA_MODE_TUNNEL,
                    .direction = RTE_SECURITY_IPSEC_SA_DIR_INGRESS,
                    .options = { 0 }
            },
            .crypto_capabilities = pmd_capabilities
    },
    { /* PDCP Lookaside Protocol offload Data Plane */
            .action = RTE_SECURITY_ACTION_TYPE_LOOKASIDE_PROTOCOL,
            .protocol = RTE_SECURITY_PROTOCOL_PDCP,
```

(continues on next page)

(continued from previous page)

```
            .pdcp = {
                    .domain = RTE_SECURITY_PDCP_MODE_DATA,
                    .capa_flags = 0
            },
            .crypto_capabilities = pmd_capabilities
    },
    { /* PDCP Lookaside Protocol offload Control */
            .action = RTE_SECURITY_ACTION_TYPE_LOOKASIDE_PROTOCOL,
            .protocol = RTE_SECURITY_PROTOCOL_PDCP,
            .pdcp = {
                    .domain = RTE_SECURITY_PDCP_MODE_CONTROL,
                    .capa_flags = 0
            },
            .crypto_capabilities = pmd_capabilities
    },
    {
            .action = RTE_SECURITY_ACTION_TYPE_NONE
    }
};
static const struct rte_cryptodev_capabilities pmd_capabilities[] = {
    {    /* SHA1 HMAC */
        .op = RTE_CRYPTO_OP_TYPE_SYMMETRIC,
        .sym = {
            .xform_type = RTE_CRYPTO_SYM_XFORM_AUTH,
            .auth = {
                .algo = RTE_CRYPTO_AUTH_SHA1_HMAC,
                .block_size = 64,
                .key_size = {
                    .min = 64,
                    .max = 64,
                    .increment = 0
                },
                .digest_size = {
                    .min = 12,
                    .max = 12,
                    .increment = 0
                },
                .aad_size = { 0 },
                .iv_size = { 0 }
            }
        }
    },
    {    /* AES CBC */
        .op = RTE_CRYPTO_OP_TYPE_SYMMETRIC,
        .sym = {
            .xform_type = RTE_CRYPTO_SYM_XFORM_CIPHER,
            .cipher = {
                .algo = RTE_CRYPTO_CIPHER_AES_CBC,
                .block_size = 16,
                .key_size = {
                    .min = 16,
                    .max = 32,
                    .increment = 8
                },
                .iv_size = {
                    .min = 16,
                    .max = 16,
                    .increment = 0
                }
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Capabilities Discovery

Discovering the features and capabilities of a driver (crypto/ethernet) is achieved through the `rte_security_capabilities_get()` function.

```
const struct rte_security_capability *rte_security_capabilities_get(uint16_t id);
```

This allows the user to query a specific driver and get all device security capabilities. It returns an array of `rte_security_capability` structures which contains all the capabilities for that device.

Security Session Create/Free

Security Sessions are created to store the immutable fields of a particular Security Association for a particular protocol which is defined by a security session configuration structure which is used in the operation processing of a packet flow. Sessions are used to manage protocol specific information as well as crypto parameters. Security sessions cache this immutable data in a optimal way for the underlying PMD and this allows further acceleration of the offload of Crypto workloads.

The Security framework provides APIs to create and free sessions for crypto/ethernet devices, where sessions are mempool objects. It is the application's responsibility to create and manage the session mempools. The mempool object size should be able to accommodate the driver's private data of security session.

Once the session mempools have been created, `rte_security_session_create()` is used to allocate and initialize a session for the required crypto/ethernet device.

Session APIs need a parameter `rte_security_ctx` to identify the crypto/ethernet security ops. This parameter can be retrieved using the APIs `rte_cryptodev_get_sec_ctx()` (for crypto device) or `rte_eth_dev_get_sec_ctx` (for ethernet port).

Sessions already created can be updated with `rte_security_session_update()`.

When a session is no longer used, the user must call `rte_security_session_destroy()` to free the driver private session data and return the memory back to the mempool.

For look aside protocol offload to hardware crypto device, the `rte_crypto_op` created by the application is attached to the security session by the API `rte_security_attach_session()`.

For Inline Crypto and Inline protocol offload, device specific defined metadata is updated in the mbuf using `rte_security_set_pkt_metadata()` if `DEV_TX_OFFLOAD_SEC_NEED_MDATA` is set.

For inline protocol offloaded ingress traffic, the application can register a pointer, `userdata` , in the security session. When the packet is received, `rte_security_get_userdata()` would return the userdata registered for the security session which processed the packet.

Note: In case of inline processed packets, `rte_mbuf.udata64` field would be used by the driver to relay information on the security processing associated with the packet. In ingress, the driver would set this in Rx path while in egress, `rte_security_set_pkt_metadata()` would perform a similar operation.

The application is expected not to modify the field when it has relevant info. For ingress, this device-specific 64 bit value is required to derive other information (like userdata), required for identifying the security processing done on the packet.

Security session configuration

Security Session configuration structure is defined as `rte_security_session_conf`

```
struct rte_security_session_conf {
    enum rte_security_session_action_type action_type;
    /**< Type of action to be performed on the session */
    enum rte_security_session_protocol protocol;
    /**< Security protocol to be configured */
    union {
            struct rte_security_ipsec_xform ipsec;
            struct rte_security_macsec_xform macsec;
            struct rte_security_pdcp_xform pdcp;
    };
    /**< Configuration parameters for security session */
    struct rte_crypto_sym_xform *crypto_xform;
    /**< Security Session Crypto Transformations */
    void *userdata;
    /**< Application specific userdata to be saved with session */
};
```

The configuration structure reuses the `rte_crypto_sym_xform` struct for crypto related configuration. The `rte_security_session_action_type` struct is used to specify whether the session is configured for Lookaside Protocol offload or Inline Crypto or Inline Protocol Offload.

```
enum rte_security_session_action_type {
    RTE_SECURITY_ACTION_TYPE_NONE,
    /**< No security actions */
    RTE_SECURITY_ACTION_TYPE_INLINE_CRYPTO,
    /**< Crypto processing for security protocol is processed inline
     * during transmission
     */
    RTE_SECURITY_ACTION_TYPE_INLINE_PROTOCOL,
    /**< All security protocol processing is performed inline during
     * transmission
     */
    RTE_SECURITY_ACTION_TYPE_LOOKASIDE_PROTOCOL,
    /**< All security protocol processing including crypto is performed
     * on a lookaside accelerator
     */
    RTE_SECURITY_ACTION_TYPE_CPU_CRYPTO
    /**< Similar to ACTION_TYPE_NONE but crypto processing for security
     * protocol is processed synchronously by a CPU.
     */
};
```

The `rte_security_session_protocol` is defined as

```
enum rte_security_session_protocol {
    RTE_SECURITY_PROTOCOL_IPSEC = 1,
    /**< IPsec Protocol */
    RTE_SECURITY_PROTOCOL_MACSEC,
    /**< MACSec Protocol */
    RTE_SECURITY_PROTOCOL_PDCP,
```

(continues on next page)

(continued from previous page)

```
    /**< PDCP Protocol */
};
```

Currently the library defines configuration parameters for IPsec and PDCP only. For other protocols like MACSec, structures and enums are defined as place holders which will be updated in the future.

IPsec related configuration parameters are defined in `rte_security_ipsec_xform`

```
struct rte_security_ipsec_xform {
    uint32_t spi;
    /**< SA security parameter index */
    uint32_t salt;
    /**< SA salt */
    struct rte_security_ipsec_sa_options options;
    /**< various SA options */
    enum rte_security_ipsec_sa_direction direction;
    /**< IPsec SA Direction - Egress/Ingress */
    enum rte_security_ipsec_sa_protocol proto;
    /**< IPsec SA Protocol - AH/ESP */
    enum rte_security_ipsec_sa_mode mode;
    /**< IPsec SA Mode - transport/tunnel */
    struct rte_security_ipsec_tunnel_param tunnel;
    /**< Tunnel parameters, NULL for transport mode */
};
```

PDCP related configuration parameters are defined in `rte_security_pdcp_xform`

```
struct rte_security_pdcp_xform {
    int8_t bearer;  /**< PDCP bearer ID */
    /** Enable in order delivery, this field shall be set only if
     * driver/HW is capable. See RTE_SECURITY_PDCP_ORDERING_CAP.
     */
    uint8_t en_ordering;
    /** Notify driver/HW to detect and remove duplicate packets.
     * This field should be set only when driver/hw is capable.
     * See RTE_SECURITY_PDCP_DUP_DETECT_CAP.
     */
    uint8_t remove_duplicates;
    /** PDCP mode of operation: Control or data */
    enum rte_security_pdcp_domain domain;
    /** PDCP Frame Direction 0:UL 1:DL */
    enum rte_security_pdcp_direction pkt_dir;
    /** Sequence number size, 5/7/12/15/18 */
    enum rte_security_pdcp_sn_size sn_size;
    /** Starting Hyper Frame Number to be used together with the SN
     * from the PDCP frames
     */
    uint32_t hfn;
    /** HFN Threshold for key renegotiation */
    uint32_t hfn_threshold;
};
```

Security API

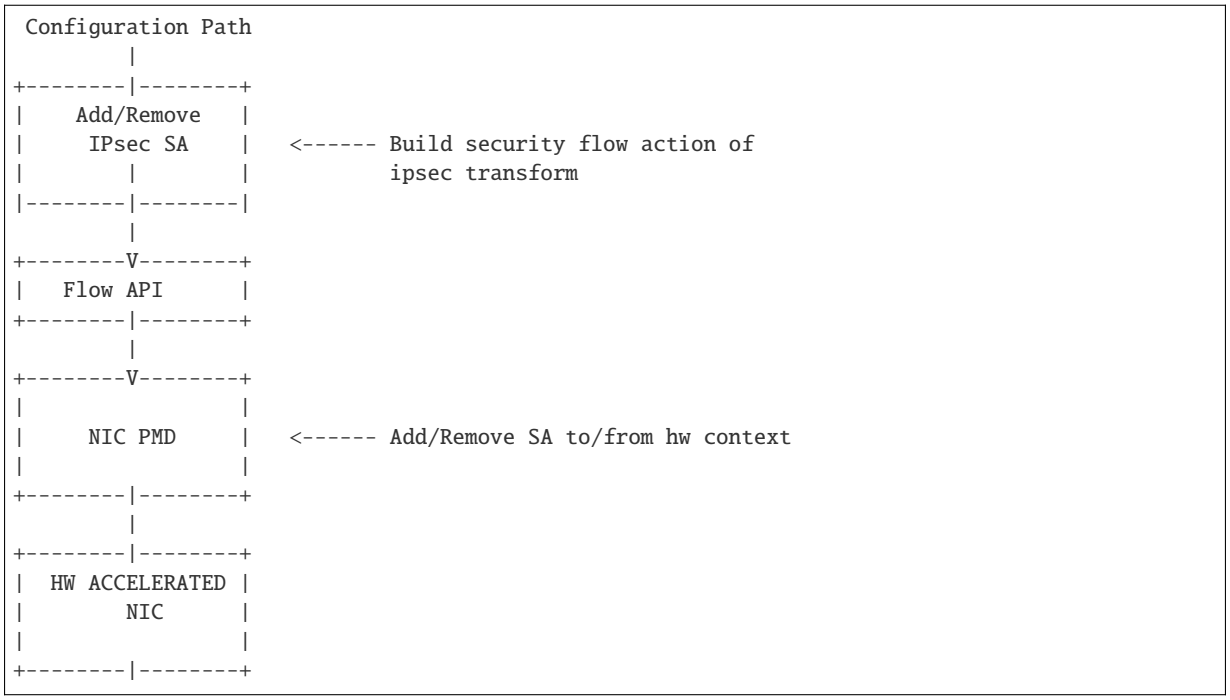
The rte_security Library API is described in the *DPDK API Reference* document.

Flow based Security Session

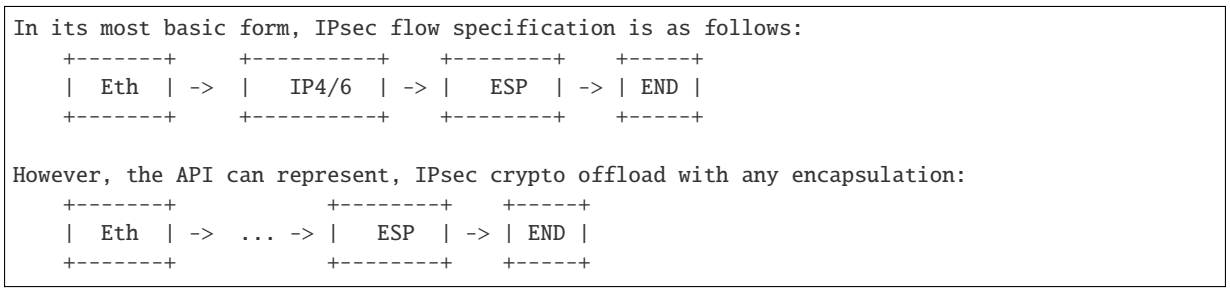
In the case of NIC based offloads, the security session specified in the 'rte_flow_action_security' must be created on the same port as the flow action that is being specified.

The ingress/egress flow attribute should match that specified in the security session if the security session supports the definition of the direction.

Multiple flows can be configured to use the same security session. For example if the security session specifies an egress IPsec SA, then multiple flows can be specified to that SA. In the case of an ingress IPsec SA then it is only valid to have a single flow to map to that security session.



- Add/Delete SA flow: To add a new inline SA construct a rte_flow_item for Ethernet + IP + ESP using the SA selectors and the `rte_crypto_ipsec_xform` as the `rte_flow_action`. Note that any rte_flow_items may be empty, which means it is not checked.



5.20 Rawdevice Library

5.20.1 Introduction

In terms of device flavor (type) support, DPDK currently has ethernet (lib_ether), cryptodev (libcryptodev), eventdev (libeventdev) and vdev (virtual device) support.

For a new type of device, for example an accelerator, there are not many options except: 1. create another lib/librte_MySpecialDev, driver/MySpecialDrv and use it through Bus/PMD model. 2. Or, create a vdev and implement necessary custom APIs which are directly exposed from driver layer. However this may still require changes in bus code in DPDK.

The DPDK Rawdev library is an abstraction that provides the DPDK framework a way to manage such devices in a generic manner without expecting changes to library or EAL for each device type. This library provides a generic set of operations and APIs for framework and Applications to use, respectively, for interfacing with such type of devices.

5.20.2 Design

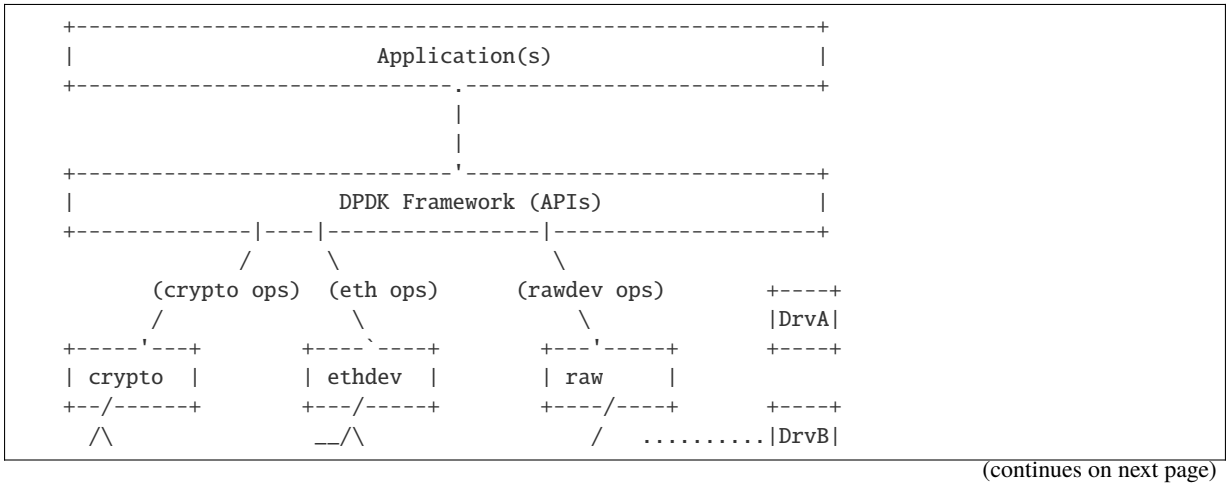
Key factors guiding design of the Rawdevice library:

1. Following are some generic operations which can be treated as applicable to a large subset of device types. None of the operations are mandatory to be implemented by a driver. Application should also be designed for proper handling for unsupported APIs.

- Device Start/Stop - In some cases, 'reset' might also be required which has different semantics than a start-stop-start cycle.
- Configuration - Device, Queue or any other sub-system configuration
- I/O - Sending a series of buffers which can enclose any arbitrary data
- Statistics - Fetch arbitrary device statistics
- Firmware Management - Firmware load/unload/status

2. Application API should be able to pass along arbitrary state information to/from device driver. This can be achieved by maintaining context information through opaque data or pointers.

Figure below outlines the layout of the rawdevice library and device vis-a-vis other well known device types like eth and crypto:



(continues on next page)

(continued from previous page)

```
    /  \             /    \               / ../    \     +----+
 +====+ +====+    +====+ +====+           +==/=+      ```Bus Probe
 |DevA| |DevB|    |DevC| |DevD|           |DevF|
 +====+ +====+    +====+ +====+           +====+
   |      |        |      |                |
 ``|``````|````````|``````|````````````````|````````Bus Scan
  (PCI)   |       (PCI)  (PCI)           (PCI)
        (BusA)

* It is assumed above that DrvB is a PCI type driver which registers itself
  with PCI Bus
* Thereafter, when the PCI scan is done, during probe DrvB would match the
  rawdev DevF ID and take control of device
* Applications can then continue using the device through rawdev API
  interfaces
```

Device Identification

Physical rawdev devices are discovered during the Bus scan executed at DPDK initialization, based on their identification and probing with corresponding driver. Thus, a generic device needs to have an identifier and a driver capable of identifying it through this identifier.

Virtual devices can be created by two mechanisms, either using the EAL command line options or from within the application using an EAL API directly.

From the command line using the –vdev EAL option

```
--vdev 'rawdev_dev1'
```

Or using the rte_vdev_init API within the application code.

```
rte_vdev_init("rawdev_dev1", NULL)
```

5.21 Link Bonding Poll Mode Driver Library

In addition to Poll Mode Drivers (PMDs) for physical and virtual hardware, DPDK also includes a pure-software library that allows physical PMDs to be bonded together to create a single logical PMD.

Fig. 5.29: Bonded PMDs

The Link Bonding PMD library(librte_pmd_bond) supports bonding of groups of `rte_eth_dev` ports of the same speed and duplex to provide similar capabilities to that found in Linux bonding driver to allow the aggregation of multiple (slave) NICs into a single logical interface between a server and a switch. The new bonded PMD will then process these interfaces based on the mode of operation specified to provide support for features such as redundant links, fault tolerance and/or load balancing.

The librte_pmd_bond library exports a C API which provides an API for the creation of bonded devices as well as the configuration and management of the bonded device and its slave devices.

Note: The Link Bonding PMD Library is enabled by default in the build configuration files, the library can be disabled by setting `CONFIG_RTE_LIBRTE_PMD_BOND=n` and recompiling the DPDK.

5.21.1 Link Bonding Modes Overview

Currently the Link Bonding PMD library supports following modes of operation:

- **Round-Robin (Mode 0):**

Fig. 5.30: Round-Robin (Mode 0)

This mode provides load balancing and fault tolerance by transmission of packets in sequential order from the first available slave device through the last. Packets are bulk dequeued from devices then serviced in a round-robin manner. This mode does not guarantee in order reception of packets and down stream should be able to handle out of order packets.

- **Active Backup (Mode 1):**

Fig. 5.31: Active Backup (Mode 1)

In this mode only one slave in the bond is active at any time, a different slave becomes active if, and only if, the primary active slave fails, thereby providing fault tolerance to slave failure. The single logical bonded interface's MAC address is externally visible on only one NIC (port) to avoid confusing the network switch.

- **Balance XOR (Mode 2):**

Fig. 5.32: Balance XOR (Mode 2)

This mode provides transmit load balancing (based on the selected transmission policy) and fault tolerance. The default policy (layer2) uses a simple calculation based on the packet flow source and destination MAC addresses as well as the number of active slaves available to the bonded device to classify the packet to a specific slave to transmit on. Alternate transmission policies supported are layer 2+3, this takes the IP source and destination addresses into the calculation of the transmit slave port and the final supported policy is layer 3+4, this uses IP source and destination addresses as well as the TCP/UDP source and destination port.

Note: The coloring differences of the packets are used to identify different flow classification calculated by the selected transmit policy

---

- **Broadcast (Mode 3):**
- **Link Aggregation 802.3AD (Mode 4):**
- **Transmit Load Balancing (Mode 5):**

Fig. 5.33: Broadcast (Mode 3)

This mode provides fault tolerance by transmission of packets on all slave ports.

Fig. 5.34: Link Aggregation 802.3AD (Mode 4)

This mode provides dynamic link aggregation according to the 802.3ad specification. It negotiates and monitors aggregation groups that share the same speed and duplex settings using the selected balance transmit policy for balancing outgoing traffic.

DPDK implementation of this mode provide some additional requirements of the application.

1. It needs to call `rte_eth_tx_burst` and `rte_eth_rx_burst` with intervals period of less than 100ms.
2. Calls to `rte_eth_tx_burst` must have a buffer size of at least 2xN, where N is the number of slaves. This is a space required for LACP frames. Additionally LACP packets are included in the statistics, but they are not returned to the application.

5.21.2 Implementation Details

The librte_pmd_bond bonded device are compatible with the Ethernet device API exported by the Ethernet PMDs described in the *DPDK API Reference*.

The Link Bonding Library supports the creation of bonded devices at application startup time during EAL initialization using the `--vdev` option as well as programmatically via the C API `rte_eth_bond_create` function.

Bonded devices support the dynamical addition and removal of slave devices using the `rte_eth_bond_slave_add` / `rte_eth_bond_slave_remove` APIs.

After a slave device is added to a bonded device slave is stopped using `rte_eth_dev_stop` and then reconfigured using `rte_eth_dev_configure` the RX and TX queues are also reconfigured using `rte_eth_tx_queue_setup` / `rte_eth_rx_queue_setup` with the parameters use to configure the bonding device. If RSS is enabled for bonding device, this mode is also enabled on new slave and configured as well. Any flow which was configured to the bond device also is configured to the added slave.

Setting up multi-queue mode for bonding device to RSS, makes it fully RSS-capable, so all slaves are synchronized with its configuration. This mode is intended to provide RSS configuration on slaves transparent for client application implementation.

Bonding device stores its own version of RSS settings i.e. RETA, RSS hash function and RSS key, used to set up its slaves. That let to define the meaning of RSS configuration of bonding device as desired configuration of whole bonding (as one unit), without pointing any of slave inside. It is required to ensure consistency and made it more error-proof.

RSS hash function set for bonding device, is a maximal set of RSS hash functions supported by all bonded slaves. RETA size is a GCD of all its RETA's sizes, so it can be easily used as a pattern providing

Fig. 5.35: Transmit Load Balancing (Mode 5)

This mode provides an adaptive transmit load balancing. It dynamically changes the transmitting slave, according to the computed load. Statistics are collected in 100ms intervals and scheduled every 10ms.

expected behavior, even if slave RETAs' sizes are different. If RSS Key is not set for bonded device, it's not changed on the slaves and default key for device is used.

As RSS configurations, there is flow consistency in the bonded slaves for the next rte flow operations:

Validate:

- Validate flow for each slave, failure at least for one slave causes to bond validation failure.

Create:

- Create the flow in all slaves.
- Save all the slaves created flows objects in bonding internal flow structure.
- Failure in flow creation for existed slave rejects the flow.
- Failure in flow creation for new slaves in slave adding time rejects the slave.

Destroy:

- Destroy the flow in all slaves and release the bond internal flow memory.

Flush:

- Destroy all the bonding PMD flows in all the slaves.

Note: Don't call slaves flush directly, It destroys all the slave flows which may include external flows or the bond internal LACP flow.

Query:

- Summarize flow counters from all the slaves, relevant only for `RTE_FLOW_ACTION_TYPE_COUNT`.

Isolate:

- Call to flow isolate for all slaves.
- Failure in flow isolation for existed slave rejects the isolate mode.
- Failure in flow isolation for new slaves in slave adding time rejects the slave.

All settings are managed through the bonding port API and always are propagated in one direction (from bonding to slaves).

Link Status Change Interrupts / Polling

Link bonding devices support the registration of a link status change callback, using the `rte_eth_dev_callback_register` API, this will be called when the status of the bonding device changes. For example in the case of a bonding device which has 3 slaves, the link status will change to up when one slave becomes active or change to down when all slaves become inactive. There is no callback notification when a single slave changes state and the previous conditions are not met. If a user wishes to monitor individual slaves then they must register callbacks with that slave directly.

The link bonding library also supports devices which do not implement link status change interrupts, this is achieved by polling the devices link status at a defined period which is set using the `rte_eth_bond_link_monitoring_set` API, the default polling interval is 10ms. When a device is

added as a slave to a bonding device it is determined using the RTE_PCI_DRV_INTR_LSC flag whether the device supports interrupts or whether the link status should be monitored by polling it.

Requirements / Limitations

The current implementation only supports devices that support the same speed and duplex to be added as a slaves to the same bonded device. The bonded device inherits these attributes from the first active slave added to the bonded device and then all further slaves added to the bonded device must support these parameters.

A bonding device must have a minimum of one slave before the bonding device itself can be started.

To use a bonding device dynamic RSS configuration feature effectively, it is also required, that all slaves should be RSS-capable and support, at least one common hash function available for each of them. Changing RSS key is only possible, when all slave devices support the same key size.

To prevent inconsistency on how slaves process packets, once a device is added to a bonding device, RSS and rte flow configurations should be managed through the bonding device API, and not directly on the slave.

Like all other PMD, all functions exported by a PMD are lock-free functions that are assumed not to be invoked in parallel on different logical cores to work on the same target object.

It should also be noted that the PMD receive function should not be invoked directly on a slave devices after they have been to a bonded device since packets read directly from the slave device will no longer be available to the bonded device to read.

Configuration

Link bonding devices are created using the rte_eth_bond_create API which requires a unique device name, the bonding mode, and the socket Id to allocate the bonding device's resources on. The other configurable parameters for a bonded device are its slave devices, its primary slave, a user defined MAC address and transmission policy to use if the device is in balance XOR mode.

Slave Devices

Bonding devices support up to a maximum of RTE_MAX_ETHPORTS slave devices of the same speed and duplex. Ethernet devices can be added as a slave to a maximum of one bonded device. Slave devices are reconfigured with the configuration of the bonded device on being added to a bonded device.

The bonded also guarantees to return the MAC address of the slave device to its original value of removal of a slave from it.

Primary Slave

The primary slave is used to define the default port to use when a bonded device is in active backup mode. A different port will only be used if, and only if, the current primary port goes down. If the user does not specify a primary port it will default to being the first port added to the bonded device.

MAC Address

The bonded device can be configured with a user specified MAC address, this address will be inherited by the some/all slave devices depending on the operating mode. If the device is in active backup mode then only the primary device will have the user specified MAC, all other slaves will retain their original MAC address. In mode 0, 2, 3, 4 all slaves devices are configure with the bonded devices MAC address.

If a user defined MAC address is not defined then the bonded device will default to using the primary slaves MAC address.

Balance XOR Transmit Policies

There are 3 supported transmission policies for bonded device running in Balance XOR mode. Layer 2, Layer 2+3, Layer 3+4.

- **Layer 2:** Ethernet MAC address based balancing is the default transmission policy for Balance XOR bonding mode. It uses a simple XOR calculation on the source MAC address and destination MAC address of the packet and then calculate the modulus of this value to calculate the slave device to transmit the packet on.
- **Layer 2 + 3:** Ethernet MAC address & IP Address based balancing uses a combination of source/destination MAC addresses and the source/destination IP addresses of the data packet to decide which slave port the packet will be transmitted on.
- **Layer 3 + 4:** IP Address & UDP Port based balancing uses a combination of source/destination IP Address and the source/destination UDP ports of the packet of the data packet to decide which slave port the packet will be transmitted on.

All these policies support 802.1Q VLAN Ethernet packets, as well as IPv4, IPv6 and UDP protocols for load balancing.

5.21.3 Using Link Bonding Devices

The librte_pmd_bond library supports two modes of device creation, the libraries export full C API or using the EAL command line to statically configure link bonding devices at application startup. Using the EAL option it is possible to use link bonding functionality transparently without specific knowledge of the libraries API, this can be used, for example, to add bonding functionality, such as active backup, to an existing application which has no knowledge of the link bonding C API.

Using the Poll Mode Driver from an Application

Using the librte_pmd_bond libraries API it is possible to dynamically create and manage link bonding device from within any application. Link bonding devices are created using the `rte_eth_bond_create` API which requires a unique device name, the link bonding mode to initial the device in and finally the socket Id which to allocate the devices resources onto. After successful creation of a bonding device it must be configured using the generic Ethernet device configure API `rte_eth_dev_configure` and then the RX and TX queues which will be used must be setup using `rte_eth_tx_queue_setup` / `rte_eth_rx_queue_setup`.

Slave devices can be dynamically added and removed from a link bonding device using the `rte_eth_bond_slave_add` / `rte_eth_bond_slave_remove` APIs but at least one slave device must be added to the link bonding device before it can be started using `rte_eth_dev_start`.

The link status of a bonded device is dictated by that of its slaves, if all slave device link status are down or if all slaves are removed from the link bonding device then the link status of the bonding device will go down.

It is also possible to configure / query the configuration of the control parameters of a bonded device using the provided APIs `rte_eth_bond_mode_set/ get`, `rte_eth_bond_primary_set/get`, `rte_eth_bond_mac_set/reset` and `rte_eth_bond_xmit_policy_set/get`.

Using Link Bonding Devices from the EAL Command Line

Link bonding devices can be created at application startup time using the `--vdev` EAL command line option. The device name must start with the net_bonding prefix followed by numbers or letters. The name must be unique for each device. Each device can have multiple options arranged in a comma separated list. Multiple devices definitions can be arranged by calling the `--vdev` option multiple times.

Device names and bonding options must be separated by commas as shown below:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bonding0,bond_opt0=..,bond opt1=..'--vdev 'net_
↪bonding1,bond _opt0=..,bond_opt1=..'
```

Link Bonding EAL Options

There are multiple ways of definitions that can be assessed and combined as long as the following two rules are respected:

- A unique device name, in the format of net_bondingX is provided, where X can be any combination of numbers and/or letters, and the name is no greater than 32 characters long.
- A least one slave device is provided with for each bonded device definition.
- The operation mode of the bonded device being created is provided.

The different options are:

- mode: Integer value defining the bonding mode of the device. Currently supports modes 0,1,2,3,4,5 (round-robin, active backup, balance, broadcast, link aggregation, transmit load balancing).

```
mode=2
```

- slave: Defines the PMD device which will be added as slave to the bonded device. This option can be selected multiple times, for each device to be added as a slave. Physical devices should be specified using their PCI address, in the format domain:bus:devid.function

```
slave=0000:0a:00.0,slave=0000:0a:00.1
```

- primary: Optional parameter which defines the primary slave port, is used in active backup mode to select the primary slave for data TX/RX if it is available. The primary port also is used to select the MAC address to use when it is not defined by the user. This defaults to the first slave added to the device if it is specified. The primary device must be a slave of the bonded device.

```
primary=0000:0a:00.0
```

- socket_id: Optional parameter used to select which socket on a NUMA device the bonded devices resources will be allocated on.

```
socket_id=0
```

- mac: Optional parameter to select a MAC address for link bonding device, this overrides the value of the primary slave device.

```
mac=00:1e:67:1d:fd:1d
```

- xmit_policy: Optional parameter which defines the transmission policy when the bonded device is in balance mode. If not user specified this defaults to l2 (layer 2) forwarding, the other transmission policies available are l23 (layer 2+3) and l34 (layer 3+4)

```
xmit_policy=l23
```

- lsc_poll_period_ms: Optional parameter which defines the polling interval in milli-seconds at which devices which don't support lsc interrupts are checked for a change in the devices link status

```
lsc_poll_period_ms=100
```

- up_delay: Optional parameter which adds a delay in milli-seconds to the propagation of a devices link status changing to up, by default this parameter is zero.

```
up_delay=10
```

- down_delay: Optional parameter which adds a delay in milli-seconds to the propagation of a devices link status changing to down, by default this parameter is zero.

```
down_delay=50
```

Examples of Usage

Create a bonded device in round robin mode with two slaves specified by their PCI address:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bonding0,mode=0,slave=0000:0a:00.01,
↪slave=0000:04:00.00' -- --port-topology=chained
```

Create a bonded device in round robin mode with two slaves specified by their PCI address and an overriding MAC address:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bonding0,mode=0,slave=0000:0a:00.01,
↪slave=0000:04:00.00,mac=00:1e:67:1d:fd:1d' -- --port-topology=chained
```

Create a bonded device in active backup mode with two slaves specified, and a primary slave specified by their PCI addresses:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bonding0,mode=1,slave=0000:0a:00.01,
↪slave=0000:04:00.00,primary=0000:0a:00.01' -- --port-topology=chained
```

Create a bonded device in balance mode with two slaves specified by their PCI addresses, and a transmission policy of layer 3 + 4 forwarding:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev 'net_bonding0,mode=2,slave=0000:0a:00.01,
↪slave=0000:04:00.00,xmit_policy=l34' -- --port-topology=chained
```

5.22 Timer Library

The Timer library provides a timer service to DPDK execution units to enable execution of callback functions asynchronously. Features of the library are:

- Timers can be periodic (multi-shot) or single (one-shot).
- Timers can be loaded from one core and executed on another. It has to be specified in the call to rte_timer_reset().
- Timers provide high precision (depends on the call frequency to rte_timer_manage() that checks timer expiration for the local core).
- If not required in the application, timers can be disabled at compilation time by not calling the rte_timer_manage() to increase performance.

The timer library uses the rte_get_timer_cycles() function that uses the High Precision Event Timer (HPET) or the CPUs Time Stamp Counter (TSC) to provide a reliable time reference.

This library provides an interface to add, delete and restart a timer. The API is based on BSD callout() with a few differences. Refer to the callout manual.

5.22.1 Implementation Details

Timers are tracked on a per-lcore basis, with all pending timers for a core being maintained in order of timer expiry in a skiplist data structure. The skiplist used has ten levels and each entry in the table appears in each level with probability ¼^level. This means that all entries are present in level 0, 1 in every 4 entries is present at level 1, one in every 16 at level 2 and so on up to level 9. This means that adding and removing entries from the timer list for a core can be done in log(n) time, up to 4^10 entries, that is, approximately 1,000,000 timers per lcore.

A timer structure contains a special field called status, which is a union of a timer state (stopped, pending, running, config) and an owner (lcore id). Depending on the timer state, we know if a timer is present in a list or not:

- STOPPED: no owner, not in a list
- CONFIG: owned by a core, must not be modified by another core, maybe in a list or not, depending on previous state

- PENDING: owned by a core, present in a list
- RUNNING: owned by a core, must not be modified by another core, present in a list

Resetting or stopping a timer while it is in a CONFIG or RUNNING state is not allowed. When modifying the state of a timer, a Compare And Swap instruction should be used to guarantee that the status (state+owner) is modified atomically.

Inside the rte_timer_manage() function, the skiplist is used as a regular list by iterating along the level 0 list, which contains all timer entries, until an entry which has not yet expired has been encountered. To improve performance in the case where there are entries in the timer list but none of those timers have yet expired, the expiry time of the first list entry is maintained within the per-core timer list structure itself. On 64-bit platforms, this value can be checked without the need to take a lock on the overall structure. (Since expiry times are maintained as 64-bit values, a check on the value cannot be done on 32-bit platforms without using either a compare-and-swap (CAS) instruction or using a lock, so this additional check is skipped in favor of checking as normal once the lock has been taken.) On both 64-bit and 32-bit platforms, a call to rte_timer_manage() returns without taking a lock in the case where the timer list for the calling core is empty.

5.22.2 Use Cases

The timer library is used for periodic calls, such as garbage collectors, or some state machines (ARP, bridging, and so on).

5.22.3 References

- callout manual - The callout facility that provides timers with a mechanism to execute a function at a given time.
- HPET - Information about the High Precision Event Timer (HPET).

5.23 Hash Library

The DPDK provides a Hash Library for creating hash table for fast lookup. The hash table is a data structure optimized for searching through a set of entries that are each identified by a unique key. For increased performance the DPDK Hash requires that all the keys have the same number of bytes which is set at the hash creation time.

5.23.1 Hash API Overview

The main configuration parameters for the hash table are:

- Total number of hash entries in the table
- Size of the key in bytes
- An extra flag to describe additional settings, for example the multithreading mode of operation and extendable bucket functionality (as will be described later)

The hash table also allows the configuration of some low-level implementation related parameters such as:

- Hash function to translate the key into a hash value

The main methods exported by the Hash Library are:

- Add entry with key: The key is provided as input. If the new entry is successfully added to the hash table for the specified key, or there is already an entry in the hash table for the specified key, then the position of the entry is returned. If the operation was not successful, for example due to lack of free entries in the hash table, then a negative value is returned.
- Delete entry with key: The key is provided as input. If an entry with the specified key is found in the hash, then the entry is removed from the hash table and the position where the entry was found in the hash table is returned. If no entry with the specified key exists in the hash table, then a negative value is returned
- Lookup for entry with key: The key is provided as input. If an entry with the specified key is found in the hash table (i.e., lookup hit), then the position of the entry is returned, otherwise (i.e., lookup miss) a negative value is returned.

Apart from the basic methods explained above, the Hash Library API provides a few more advanced methods to query and update the hash table:

- Add / lookup / delete entry with key and precomputed hash: Both the key and its precomputed hash are provided as input. This allows the user to perform these operations faster, as the hash value is already computed.
- Add / lookup entry with key and data: A data is provided as input for add. Add allows the user to store not only the key, but also the data which may be either a 8-byte integer or a pointer to external data (if data size is more than 8 bytes).
- Combination of the two options above: User can provide key, precomputed hash, and data.
- Ability to not free the position of the entry in the hash table upon calling delete. This is useful for multi-threaded scenarios where readers continue to use the position even after the entry is deleted.

Also, the API contains a method to allow the user to look up entries in batches, achieving higher performance than looking up individual entries, as the function prefetches next entries at the time it is operating with the current ones, which reduces significantly the performance overhead of the necessary memory accesses.

The actual data associated with each key can be either managed by the user using a separate table that mirrors the hash in terms of number of entries and position of each entry, as shown in the Flow Classification use case described in the following sections, or stored in the hash table itself.

The example hash tables in the L2/L3 Forwarding sample applications define which port to forward a packet to based on a packet flow identified by the five-tuple lookup. However, this table could also be used for more sophisticated features and provide many other functions and actions that could be performed on the packets and flows.

5.23.2 Multi-process support

The hash library can be used in a multi-process environment. The only function that can only be used in single-process mode is rte_hash_set_cmp_func(), which sets up a custom compare function, which is assigned to a function pointer (therefore, it is not supported in multi-process mode).

5.23.3 Multi-thread support

The hash library supports multithreading, and the user specifies the needed mode of operation at the creation time of the hash table by appropriately setting the flag. In all modes of operation lookups are thread-safe meaning lookups can be called from multiple threads concurrently.

For concurrent writes, and concurrent reads and writes the following flag values define the corresponding modes of operation:

- If the multi-writer flag (RTE_HASH_EXTRA_FLAGS_MULTI_WRITER_ADD) is set, multiple threads writing to the table is allowed. Key add, delete, and table reset are protected from other writer threads. With only this flag set, readers are not protected from ongoing writes.
- If the read/write concurrency (RTE_HASH_EXTRA_FLAGS_RW_CONCURRENCY) is set, multithread read/write operation is safe (i.e., application does not need to stop the readers from accessing the hash table until writers finish their updates. Readers and writers can operate on the table concurrently). The library uses a reader-writer lock to provide the concurrency.
- In addition to these two flag values, if the transactional memory flag (RTE_HASH_EXTRA_FLAGS_TRANS_MEM_SUPPORT) is also set, the reader-writer lock will use hardware transactional memory (e.g., Intel® TSX) if supported to guarantee thread safety. If the platform supports Intel® TSX, it is advised to set the transactional memory flag, as this will speed up concurrent table operations. Otherwise concurrent operations will be slower because of the overhead associated with the software locking mechanisms.
- If lock free read/write concurrency (RTE_HASH_EXTRA_FLAGS_RW_CONCURRENCY_LF) is set, read/write concurrency is provided without using reader-writer lock. For platforms (e.g., current ARM based platforms) that do not support transactional memory, it is advised to set this flag to achieve greater scalability in performance. If this flag is set, the (RTE_HASH_EXTRA_FLAGS_NO_FREE_ON_DEL) flag is set by default.
- If the 'do not free on delete' (RTE_HASH_EXTRA_FLAGS_NO_FREE_ON_DEL) flag is set, the position of the entry in the hash table is not freed upon calling delete(). This flag is enabled by default when the lock free read/write concurrency flag is set. The application should free the position after all the readers have stopped referencing the position. Where required, the application can make use of RCU mechanisms to determine when the readers have stopped referencing the position.

5.23.4 Extendable Bucket Functionality support

An extra flag is used to enable this functionality (flag is not set by default). When the (RTE_HASH_EXTRA_FLAGS_EXT_TABLE) is set and in the very unlikely case due to excessive hash collisions that a key has failed to be inserted, the hash table bucket is extended with a linked list to insert these failed keys. This feature is important for the workloads (e.g. telco workloads) that need to insert up to 100% of the hash table size and can't tolerate any key insertion failure (even if very few). Please note that with the 'lock free read/write concurrency' flag enabled, users need to call 'rte_hash_free_key_with_position' API in order to free the empty buckets and deleted keys, to maintain the 100% capacity guarantee.

5.23.5 Implementation Details (non Extendable Bucket Case)

The hash table has two main tables:

- First table is an array of buckets each of which consists of multiple entries, Each entry contains the signature of a given key (explained below), and an index to the second table.
- The second table is an array of all the keys stored in the hash table and its data associated to each key.

The hash library uses the Cuckoo Hash algorithm to resolve collisions. For any input key, there are two possible buckets (primary and secondary/alternative location) to store that key in the hash table, therefore only the entries within those two buckets need to be examined when the key is looked up. The Hash Library uses a hash function (configurable) to translate the input key into a 4-byte hash value. The bucket index and a 2-byte signature is derived from the hash value using partial-key hashing [partial-key].

Once the buckets are identified, the scope of the key add, delete, and lookup operations is reduced to the entries in those buckets (it is very likely that entries are in the primary bucket).

To speed up the search logic within the bucket, each hash entry stores the 2-byte key signature together with the full key for each hash table entry. For large key sizes, comparing the input key against a key from the bucket can take significantly more time than comparing the 2-byte signature of the input key against the signature of a key from the bucket. Therefore, the signature comparison is done first and the full key comparison is done only when the signatures matches. The full key comparison is still necessary, as two input keys from the same bucket can still potentially have the same 2-byte signature, although this event is relatively rare for hash functions providing good uniform distributions for the set of input keys.

Example of lookup:

First of all, the primary bucket is identified and entry is likely to be stored there. If signature was stored there, we compare its key against the one provided and return the position where it was stored and/or the data associated to that key if there is a match. If signature is not in the primary bucket, the secondary bucket is looked up, where same procedure is carried out. If there is no match there either, key is not in the table and a negative value will be returned.

Example of addition:

Like lookup, the primary and secondary buckets are identified. If there is an empty entry in the primary bucket, a signature is stored in that entry, key and data (if any) are added to the second table and the index in the second table is stored in the entry of the first table. If there is no space in the primary bucket, one of the entries on that bucket is pushed to its alternative location, and the key to be added is inserted in its position. To know where the alternative bucket of the evicted entry is, a mechanism called partial-key hashing [partial-key] is used. If there is room in the alternative bucket, the evicted entry is stored in it. If not, same process is repeated (one of the entries gets pushed) until an empty entry is found. Notice

that despite all the entry movement in the first table, the second table is not touched, which would impact greatly in performance.

In the very unlikely event that an empty entry cannot be found after certain number of displacements, key is considered not able to be added (unless extendable bucket flag is set, and in that case the bucket is extended to insert the key, as will be explained later). With random keys, this method allows the user to get more than 90% table utilization, without having to drop any stored entry (e.g. using a LRU replacement policy) or allocate more memory (extendable buckets or rehashing).

Example of deletion:

Similar to lookup, the key is searched in its primary and secondary buckets. If the key is found, the entry is marked as empty. If the hash table was configured with 'no free on delete' or 'lock free read/write concurrency', the position of the key is not freed. It is the responsibility of the user to free the position after readers are not referencing the position anymore.

5.23.6 Implementation Details (with Extendable Bucket)

When the RTE_HASH_EXTRA_FLAGS_EXT_TABLE flag is set, the hash table implementation still uses the same Cuckoo Hash algorithm to store the keys into the first and second tables. However, in the very unlikely event that a key can't be inserted after certain number of the Cuckoo displacements is reached, the secondary bucket of this key is extended with a linked list of extra buckets and the key is stored in this linked list.

In case of lookup for a certain key, as before, the primary bucket is searched for a match and then the secondary bucket is looked up. If there is no match there either, the extendable buckets (linked list of extra buckets) are searched one by one for a possible match and if there is no match the key is considered not to be in the table.

The deletion is the same as the case when the RTE_HASH_EXTRA_FLAGS_EXT_TABLE flag is not set. With one exception, if a key is deleted from any bucket and an empty location is created, the last entry from the extendable buckets associated with this bucket is displaced into this empty location to possibly shorten the linked list.

5.23.7 Entry distribution in hash table

As mentioned above, Cuckoo hash implementation pushes elements out of their bucket, if there is a new entry to be added which primary location coincides with their current bucket, being pushed to their alternative location. Therefore, as user adds more entries to the hash table, distribution of the hash values in the buckets will change, being most of them in their primary location and a few in their secondary location, which the later will increase, as table gets busier. This information is quite useful, as performance may be lower as more entries are evicted to their secondary location.

See the tables below showing example entry distribution as table utilization increases.

Table 5.94: Entry distribution measured with an example table with 1024 random entries using jhash algorithm

% Table used	% In Primary location	% In Secondary location
25	100	0
50	96.1	3.9
75	88.2	11.8
80	86.3	13.7
85	83.1	16.9
90	77.3	22.7
95.8	64.5	35.5

Table 5.95: Entry distribution measured with an example table with 1 million random entries using jhash algorithm

% Table used	% In Primary location	% In Secondary location
50	96	4
75	86.9	13.1
80	83.9	16.1
85	80.1	19.9
90	74.8	25.2
94.5	67.4	32.6

Note: Last values on the tables above are the average maximum table utilization with random keys and using Jenkins hash function.

5.23.8 Use Case: Flow Classification

Flow classification is used to map each input packet to the connection/flow it belongs to. This operation is necessary as the processing of each input packet is usually done in the context of their connection, so the same set of operations is applied to all the packets from the same flow.

Applications using flow classification typically have a flow table to manage, with each separate flow having an entry associated with it in this table. The size of the flow table entry is application specific, with typical values of 4, 16, 32 or 64 bytes.

Each application using flow classification typically has a mechanism defined to uniquely identify a flow based on a number of fields read from the input packet that make up the flow key. One example is to use the DiffServ 5-tuple made up of the following fields of the IP and transport layer packet headers: Source IP Address, Destination IP Address, Protocol, Source Port, Destination Port.

The DPDK hash provides a generic method to implement an application specific flow classification mechanism. Given a flow table implemented as an array, the application should create a hash object with the same number of entries as the flow table and with the hash key size set to the number of bytes in the selected flow key.

The flow table operations on the application side are described below:

- Add flow: Add the flow key to hash. If the returned position is valid, use it to access the flow entry in the flow table for adding a new flow or updating the information associated with an existing flow. Otherwise, the flow addition failed, for example due to lack of free entries for storing new flows.
- Delete flow: Delete the flow key from the hash. If the returned position is valid, use it to access the flow entry in the flow table to invalidate the information associated with the flow.
- Free flow: Free flow key position. If 'no free on delete' or 'lock-free read/write concurrency' flags are set, wait till the readers are not referencing the position returned during add/delete flow and then free the position. RCU mechanisms can be used to find out when the readers are not referencing the position anymore.
- Lookup flow: Lookup for the flow key in the hash. If the returned position is valid (flow lookup hit), use the returned position to access the flow entry in the flow table. Otherwise (flow lookup miss) there is no flow registered for the current packet.

5.23.9 References

- Donald E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition), 1998, Addison-Wesley Professional
- [partial-key] Bin Fan, David G. Andersen, and Michael Kaminsky, MemC3: compact and concurrent MemCache with dumber caching and smarter hashing, 2013, NSDI

5.24 Elastic Flow Distributor Library

5.24.1 Introduction

In Data Centers today, clustering and scheduling of distributed workloads is a very common task. Many workloads require a deterministic partitioning of a flat key space among a cluster of machines. When a packet enters the cluster, the ingress node will direct the packet to its handling node. For example, data-centers with disaggregated storage use storage metadata tables to forward I/O requests to the correct back end storage cluster, stateful packet inspection will use match incoming flows to signatures in flow tables to send incoming packets to their intended deep packet inspection (DPI) devices, and so on.

EFD is a distributor library that uses perfect hashing to determine a target/value for a given incoming flow key. It has the following advantages: first, because it uses perfect hashing it does not store the key itself and hence lookup performance is not dependent on the key size. Second, the target/value can be any arbitrary value hence the system designer and/or operator can better optimize service rates and inter-cluster network traffic locating. Third, since the storage requirement is much smaller than a hash-based flow table (i.e. better fit for CPU cache), EFD can scale to millions of flow keys. Finally, with the current optimized library implementation, performance is fully scalable with any number of CPU cores.

5.24.2 Flow Based Distribution

Computation Based Schemes

Flow distribution and/or load balancing can be simply done using a stateless computation, for instance using round-robin or a simple computation based on the flow key as an input. For example, a hash function can be used to direct a certain flow to a target based on the flow key (e.g. `h(key) mod n`) where h(key) is the hash value of the flow key and n is the number of possible targets.

Fig. 5.36: Load Balancing Using Front End Node

In this scheme (Fig. 5.36), the front end server/distributor/load balancer extracts the flow key from the input packet and applies a computation to determine where this flow should be directed. Intuitively, this scheme is very simple and requires no state to be kept at the front end node, and hence, storage requirements are minimum.

Fig. 5.37: Consistent Hashing

A widely used flow distributor that belongs to the same category of computation-based schemes is `consistent hashing`, shown in Fig. 5.37. Target destinations (shown in red) are hashed into the same space as the flow keys (shown in blue), and keys are mapped to the nearest target in a clockwise fashion. Dynamically adding and removing targets with consistent hashing requires only K/n keys to be remapped on average, where K is the number of keys, and n is the number of targets. In contrast, in a traditional hash-based scheme, a change in the number of targets causes nearly all keys to be remapped.

Although computation-based schemes are simple and need very little storage requirement, they suffer from the drawback that the system designer/operator can't fully control the target to assign a specific key, as this is dictated by the hash function. Deterministically co-locating of keys together (for example, to minimize inter-server traffic or to optimize for network traffic conditions, target load, etc.) is simply not possible.

Flow-Table Based Schemes

When using a Flow-Table based scheme to handle flow distribution/load balancing, in contrast with computation-based schemes, the system designer has the flexibility of assigning a given flow to any given target. The flow table (e.g. DPDK RTE Hash Library) will simply store both the flow key and the target value.

Fig. 5.38: Table Based Flow Distribution

As shown in Fig. 5.38, when doing a lookup, the flow-table is indexed with the hash of the flow key and the keys (more than one is possible, because of hash collision) stored in this index and corresponding values are retrieved. The retrieved key(s) is matched with the input flow key and if there is a match the value (target id) is returned.

The drawback of using a hash table for flow distribution/load balancing is the storage requirement, since the flow table need to store keys, signatures and target values. This doesn't allow this scheme to scale to millions of flow keys. Large tables will usually not fit in the CPU cache, and hence, the lookup performance is degraded because of the latency to access the main memory.

EFD Based Scheme

EFD combines the advantages of both flow-table based and computation-based schemes. It doesn't require the large storage necessary for flow-table based schemes (because EFD doesn't store the key as explained below), and it supports any arbitrary value for any given key.

Fig. 5.39: Searching for Perfect Hash Function

The basic idea of EFD is when a given key is to be inserted, a family of hash functions is searched until the correct hash function that maps the input key to the correct value is found, as shown in Fig. 5.39. However, rather than explicitly storing all keys and their associated values, EFD stores only indices of hash functions that map keys to values, and thereby consumes much less space than conventional flow-based tables. The lookup operation is very simple, similar to a computational-based scheme: given an input key the lookup operation is reduced to hashing that key with the correct hash function.

Fig. 5.40: Divide and Conquer for Millions of Keys

Intuitively, finding a hash function that maps each of a large number (millions) of input keys to the correct output value is effectively impossible, as a result EFD, as shown in Fig. 5.40, breaks the problem into smaller pieces (divide and conquer). EFD divides the entire input key set into many small groups. Each group consists of approximately 20-28 keys (a configurable parameter for the library), then, for each small group, a brute force search to find a hash function that produces the correct outputs for each key in the group.

It should be mentioned that, since the online lookup table for EFD doesn't store the key itself, the size of the EFD table is independent of the key size and hence EFD lookup performance which is almost constant irrespective of the length of the key which is a highly desirable feature especially for longer keys.

In summary, EFD is a set separation data structure that supports millions of keys. It is used to distribute a given key to an intended target. By itself EFD is not a FIB data structure with an exact match the input flow key.

5.24.3 Example of EFD Library Usage

EFD can be used along the data path of many network functions and middleboxes. As previously mentioned, it can used as an index table for <key,value> pairs, meta-data for objects, a flow-level load balancer, etc. Fig. 5.41 shows an example of using EFD as a flow-level load balancer, where flows are received at a front end server before being forwarded to the target back end server for processing. The system designer would deterministically co-locate flows together in order to minimize cross-server interaction. (For example, flows requesting certain webpage objects are co-located together, to minimize forwarding of common objects across servers).

Fig. 5.41: EFD as a Flow-Level Load Balancer

As shown in Fig. 5.41, the front end server will have an EFD table that stores for each group what is the perfect hash index that satisfies the correct output. Because the table size is small and fits in cache (since keys are not stored), it sustains a large number of flows (N*X, where N is the maximum number of flows served by each back end server of the X possible targets).

With an input flow key, the group id is computed (for example, using last few bits of CRC hash) and then the EFD table is indexed with the group id to retrieve the corresponding hash index to use. Once the

index is retrieved the key is hashed using this hash function and the result will be the intended correct target where this flow is supposed to be processed.

It should be noted that as a result of EFD not matching the exact key but rather distributing the flows to a target back end node based on the perfect hash index, a key that has not been inserted before will be distributed to a valid target. Hence, a local table which stores the flows served at each node is used and is exact matched with the input key to rule out new never seen before flows.

5.24.4 Library API Overview

The EFD library API is created with a very similar semantics of a hash-index or a flow table. The application creates an EFD table for a given maximum number of flows, a function is called to insert a flow key with a specific target value, and another function is used to retrieve target values for a given individual flow key or a bulk of keys.

EFD Table Create

The function `rte_efd_create()` is used to create and return a pointer to an EFD table that is sized to hold up to num_flows key. The online version of the EFD table (the one that does not store the keys and is used for lookups) will be allocated and created in the last level cache (LLC) of the socket defined by the online_socket_bitmask, while the offline EFD table (the one that stores the keys and is used for key inserts and for computing the perfect hashing) is allocated and created in the LLC of the socket defined by offline_socket_bitmask. It should be noted, that for highest performance the socket id should match that where the thread is running, i.e. the online EFD lookup table should be created on the same socket as where the lookup thread is running.

EFD Insert and Update

The EFD function to insert a key or update a key to a new value is `rte_efd_update()`. This function will update an existing key to a new value (target) if the key has already been inserted before, or will insert the <key,value> pair if this key has not been inserted before. It will return 0 upon success. It will return `EFD_UPDATE_WARN_GROUP_FULL (1)` if the operation is insert, and the last available space in the key's group was just used. It will return `EFD_UPDATE_FAILED (2)` when the insertion or update has failed (either it failed to find a suitable perfect hash or the group was full). The function will return `EFD_UPDATE_NO_CHANGE (3)` if there is no change to the EFD table (i.e, same value already exists).

Note: This function is not multi-thread safe and should only be called from one thread.

EFD Lookup

To lookup a certain key in an EFD table, the function `rte_efd_lookup()` is used to return the value associated with single key. As previously mentioned, if the key has been inserted, the correct value inserted is returned, if the key has not been inserted before, a 'random' value (based on hashing of the key) is returned. For better performance and to decrease the overhead of function calls per key, it is always recommended to use a bulk lookup function (simultaneous lookup of multiple keys) instead of a single key lookup function. `rte_efd_lookup_bulk()` is the bulk lookup function, that looks up num_keys simultaneously stored in the key_list and the corresponding return values will be returned in the value_list.

Note: This function is multi-thread safe, but there should not be other threads writing in the EFD table, unless locks are used.

EFD Delete

To delete a certain key in an EFD table, the function `rte_efd_delete()` can be used. The function returns zero upon success when the key has been found and deleted. Socket_id is the parameter to use to lookup the existing value, which is ideally the caller's socket id. The previous value associated with this key will be returned in the prev_value argument.

Note: This function is not multi-thread safe and should only be called from one thread.

5.24.5 Library Internals

This section provides the brief high-level idea and an overview of the library internals to accompany the RFC. The intent of this section is to explain to readers the high-level implementation of insert, lookup and group rebalancing in the EFD library.

Insert Function Internals

As previously mentioned the EFD divides the whole set of keys into groups of a manageable size (e.g. 28 keys) and then searches for the perfect hash that satisfies the intended target value for each key. EFD stores two version of the <key,value> table:

- Offline Version (in memory): Only used for the insertion/update operation, which is less frequent than the lookup operation. In the offline version the exact keys for each group is stored. When a new key is added, the hash function is updated that will satisfy the value for the new key together with the all old keys already inserted in this group.
- Online Version (in cache): Used for the frequent lookup operation. In the online version, as previously mentioned, the keys are not stored but rather only the hash index for each group.

Fig. 5.42: Group Assignment

Fig. 5.42 depicts the group assignment for 7 flow keys as an example. Given a flow key, a hash function (in our implementation CRC hash) is used to get the group id. As shown in the figure, the groups can be unbalanced. (We highlight group rebalancing further below).

Fig. 5.43: Perfect Hash Search - Assigned Keys & Target Value

Focusing on one group that has four keys, Fig. 5.43 depicts the search algorithm to find the perfect hash function. Assuming that the target value bit for the keys is as shown in the figure, then the online EFD table will store a 16 bit hash index and 16 bit lookup table per group per value bit.

For a given keyX, a hash function `(h(keyX, seed1) + index * h(keyX, seed2))` is used to point to certain bit index in the 16bit lookup_table value, as shown in Fig. 5.44. The insert function will brute force search for all possible values for the hash index until a non conflicting lookup_table is found.

Fig. 5.44: Perfect Hash Search - Satisfy Target Values

Fig. 5.45: Finding Hash Index for Conflict Free lookup_table

For example, since both key3 and key7 have a target bit value of 1, it is okay if the hash function of both keys point to the same bit in the lookup table. A conflict will occur if a hash index is used that maps both Key4 and Key7 to the same index in the lookup_table, as shown in Fig. 5.45, since their target value bit are not the same. Once a hash index is found that produces a lookup_table with no contradictions, this index is stored for this group. This procedure is repeated for each bit of target value.

Lookup Function Internals

The design principle of EFD is that lookups are much more frequent than inserts, and hence, EFD's design optimizes for the lookups which are faster and much simpler than the slower insert procedure (inserts are slow, because of perfect hash search as previously discussed).

Fig. 5.46: EFD Lookup Operation

Fig. 5.46 depicts the lookup operation for EFD. Given an input key, the group id is computed (using CRC hash) and then the hash index for this group is retrieved from the EFD table. Using the retrieved hash index, the hash function `h(key, seed1) + index *h(key, seed2)` is used which will result in an index in the lookup_table, the bit corresponding to this index will be the target value bit. This procedure is repeated for each bit of the target value.

Group Rebalancing Function Internals

When discussing EFD inserts and lookups, the discussion is simplified by assuming that a group id is simply a result of hash function. However, since hashing in general is not perfect and will not always produce a uniform output, this simplified assumption will lead to unbalanced groups, i.e., some group will have more keys than other groups. Typically, and to minimize insert time with an increasing number of keys, it is preferable that all groups will have a balanced number of keys, so the brute force search for the perfect hash terminates with a valid hash index. In order to achieve this target, groups are rebalanced during runtime inserts, and keys are moved around from a busy group to a less crowded group as the more keys are inserted.

Fig. 5.47 depicts the high level idea of group rebalancing, given an input key the hash result is split into two parts a chunk id and 8-bit bin id. A chunk contains 64 different groups and 256 bins (i.e. for any given bin it can map to 4 distinct groups). When a key is inserted, the bin id is computed, for example in Fig. 5.47 bin_id=2, and since each bin can be mapped to one of four different groups (2 bit storage), the four possible mappings are evaluated and the one that will result in a balanced key distribution across these four is selected the mapping result is stored in these two bits.

Fig. 5.47: Runtime Group Rebalancing

5.24.6 References

1- EFD is based on collaborative research work between Intel and Carnegie Mellon University (CMU), interested readers can refer to the paper "Scaling Up Clustered Network Appliances with ScaleBricks" Dong Zhou et al. at SIGCOMM 2015 (*http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p241.pdf*) for more information.

5.25 Membership Library

5.25.1 Introduction

The DPDK Membership Library provides an API for DPDK applications to insert a new member, delete an existing member, or query the existence of a member in a given set, or a group of sets. For the case of a group of sets, the library will return not only whether the element has been inserted before in one of the sets but also which set it belongs to. The Membership Library is an extension and generalization of a traditional filter structure (for example Bloom Filter [Member-bloom]) that has multiple usages in a wide variety of workloads and applications. In general, the Membership Library is a data structure that provides a "set-summary" on whether a member belongs to a set, and as discussed in detail later, there are two advantages of using such a set-summary rather than operating on a "full-blown" complete list of elements: first, it has a much smaller storage requirement than storing the whole list of elements themselves, and secondly checking an element membership (or other operations) in this set-summary is much faster than checking it for the original full-blown complete list of elements.

We use the term "Set-Summary" in this guide to refer to the space-efficient, probabilistic membership data structure that is provided by the library. A membership test for an element will return the set this element belongs to or that the element is "not-found" with very high probability of accuracy. Set-summary is a fundamental data aggregation component that can be used in many network (and other) applications. It is a crucial structure to address performance and scalability issues of diverse network applications including overlay networks, data-centric networks, flow table summaries, network statistics and traffic monitoring. A set-summary is useful for applications who need to include a list of elements while a complete list requires too much space and/or too much processing cost. In these situations, the set-summary works as a lossy hash-based representation of a set of members. It can dramatically reduce space requirement and significantly improve the performance of set membership queries at the cost of introducing a very small membership test error probability.

Fig. 5.48: Example Usages of Membership Library

There are various usages for a Membership Library in a very large set of applications and workloads. Interested readers can refer to [Member-survey] for a survey of possible networking usages. The above figure provide a small set of examples of using the Membership Library:

- Sub-figure (a) depicts a distributed web cache architecture where a collection of proxies attempt to share their web caches (cached from a set of back-end web servers) to provide faster responses to clients, and the proxies use the Membership Library to share summaries of what web pages/objects they are caching. With the Membership Library, a proxy receiving an http request will inquire the set-summary to find its location and quickly determine whether to retrieve the requested web page from a nearby proxy or from a back-end web server.

- Sub-figure (b) depicts another example for using the Membership Library to prevent routing loops which is typically done using slow TTL countdown and dropping packets when TTL expires. As shown in Sub-figure (b), an embedded set-summary in the packet header itself can be used to summarize the set of nodes a packet has gone through, and each node upon receiving a packet can check whether its id is a member of the set of visited nodes, and if it is, then a routing loop is detected.
- Sub-Figure (c) presents another usage of the Membership Library to load-balance flows to worker threads with in-order guarantee where a set-summary is used to query if a packet belongs to an existing flow or a new flow. Packets belonging to a new flow are forwarded to the current least loaded worker thread, while those belonging to an existing flow are forwarded to the pre-assigned thread to guarantee in-order processing.
- Sub-figure (d) highlights yet another usage example in the database domain where a set-summary is used to determine joins between sets instead of creating a join by comparing each element of a set against the other elements in a different set, a join is done on the summaries since they can efficiently encode members of a given set.

Membership Library is a configurable library that is optimized to cover set membership functionality for both a single set and multi-set scenarios. Two set-summary schemes are presented including (a) vector of Bloom Filters and (b) Hash-Table based set-summary schemes with and without false negative probability. This guide first briefly describes these different types of set-summaries, usage examples for each, and then it highlights the Membership Library API.

5.25.2 Vector of Bloom Filters

Bloom Filter (BF) [Member-bloom] is a well-known space-efficient probabilistic data structure that answers set membership queries (test whether an element is a member of a set) with some probability of false positives and zero false negatives; a query for an element returns either it is "possibly in a set" (with very high probability) or "definitely not in a set".

The BF is a method for representing a set of `n` elements (for example flow keys in network applications domain) to support membership queries. The idea of BF is to allocate a bit-vector `v` with `m` bits, which are initially all set to 0. Then it chooses `k` independent hash functions `h1`, `h2`, ... `hk` with hash values range from `0` to `m-1` to perform hashing calculations on each element to be inserted. Every time when an element `X` being inserted into the set, the bits at positions `h1(X)`, `h2(X)`, ... `hk(X)` in `v` are set to 1 (any particular bit might be set to 1 multiple times for multiple different inserted elements). Given a query for any element `Y`, the bits at positions `h1(Y)`, `h2(Y)`, ... `hk(Y)` are checked. If any of them is 0, then Y is definitely not in the set. Otherwise there is a high probability that Y is a member of the set with certain false positive probability. As shown in the next equation, the false positive probability can be made arbitrarily small by changing the number of hash functions (`k`) and the vector length (`m`).

Fig. 5.49: Bloom Filter False Positive Probability

Without BF, an accurate membership testing could involve a costly hash table lookup and full element comparison. The advantage of using a BF is to simplify the membership test into a series of hash calculations and memory accesses for a small bit-vector, which can be easily optimized. Hence the lookup throughput (set membership test) can be significantly faster than a normal hash table lookup with element comparison.

Fig. 5.50: Detecting Routing Loops Using BF

BF is used for applications that need only one set, and the membership of elements is checked against the BF. The example discussed in the above figure is one example of potential applications that uses only one set to capture the node IDs that have been visited so far by the packet. Each node will then check this embedded BF in the packet header for its own id, and if the BF indicates that the current node is definitely not in the set then a loop-free route is guaranteed.

Fig. 5.51: Vector Bloom Filter (vBF) Overview

To support membership test for both multiple sets and a single set, the library implements a Vector Bloom Filter (vBF) scheme. vBF basically composes multiple bloom filters into a vector of bloom filers. The membership test is conducted on all of the bloom filters concurrently to determine which set(s) it belongs to or none of them. The basic idea of vBF is shown in the above figure where an element is used to address multiple bloom filters concurrently and the bloom filter index(es) with a hit is returned.

Fig. 5.52: vBF for Flow Scheduling to Worker Thread

As previously mentioned, there are many usages of such structures. vBF is used for applications that need to check membership against multiple sets simultaneously. The example shown in the above figure uses a set to capture all flows being assigned for processing at a given worker thread. Upon receiving a packet the vBF is used to quickly figure out if this packet belongs to a new flow so as to be forwarded to the current least loaded worker thread, or otherwise it should be queued for an existing thread to guarantee in-order processing (i.e. the property of vBF to indicate right away that a given flow is a new one or not is critical to minimize response time latency).

It should be noted that vBF can be implemented using a set of single bloom filters with sequential lookup of each BF. However, being able to concurrently search all set-summaries is a big throughput advantage. In the library, certain parallelism is realized by the implementation of checking all bloom filters together.

5.25.3 Hash-Table based Set-Summaries

Hash-table based set-summary (HTSS) is another scheme in the membership library. Cuckoo filter [Member-cfilter] is an example of HTSS. HTSS supports multi-set membership testing like vBF does. However, while vBF is better for a small number of targets, HTSS is more suitable and can easily outperform vBF when the number of sets is large, since HTSS uses a single hash table for membership testing while vBF requires testing a series of Bloom Filters each corresponding to one set. As a result, generally speaking vBF is more adequate for the case of a small limited number of sets while HTSS should be used with a larger number of sets.

Fig. 5.53: Using HTSS for Attack Signature Matching

As shown in the above figure, attack signature matching where each set represents a certain signature length (for correctness of this example, an attack signature should not be a subset of another one) in the payload is a good example for using HTSS with 0% false negative (i.e., when an element returns not found, it has a 100% certainty that it is not a member of any set). The packet inspection application benefits from knowing right away that the current payload does not match any attack signatures in the database to establish its legitimacy, otherwise a deep inspection of the packet is needed.

HTSS employs a similar but simpler data structure to a traditional hash table, and the major difference is that HTSS stores only the signatures but not the full keys/elements which can significantly reduce the footprint of the table. Along with the signature, HTSS also stores a value to indicate the target set. When

looking up an element, the element is hashed and the HTSS is addressed to retrieve the signature stored. If the signature matches then the value is retrieved corresponding to the index of the target set which the element belongs to. Because signatures can collide, HTSS can still has false positive probability. Furthermore, if elements are allowed to be overwritten or evicted when the hash table becomes full, it will also have a false negative probability. We discuss this case in the next section.

Set-Summaries with False Negative Probability

As previously mentioned, traditional set-summaries (e.g. Bloom Filters) do not have a false negative probability, i.e., it is 100% certain when an element returns "not to be present" for a given set. However, the Membership Library also supports a set-summary probabilistic data structure based on HTSS which allows for false negative probability.

In HTSS, when the hash table becomes full, keys/elements will fail to be added into the table and the hash table has to be resized to accommodate for these new elements, which can be expensive. However, if we allow new elements to overwrite or evict existing elements (as a cache typically does), then the resulting set-summary will begin to have false negative probability. This is because the element that was evicted from the set-summary may still be present in the target set. For subsequent inquiries the set-summary will falsely report the element not being in the set, hence having a false negative probability.

The major usage of HTSS with false negative is to use it as a cache for distributing elements to different target sets. By allowing HTSS to evict old elements, the set-summary can keep track of the most recent elements (i.e. active) as a cache typically does. Old inactive elements (infrequently used elements) will automatically and eventually get evicted from the set-summary. It is worth noting that the set-summary still has false positive probability, which means the application either can tolerate certain false positive or it has fall-back path when false positive happens.

Fig. 5.54: Using HTSS with False Negatives for Wild Card Classification

HTSS with false negative (i.e. a cache) also has its wide set of applications. For example wild card flow classification (e.g. ACL rules) highlighted in the above figure is an example of such application. In that case each target set represents a sub-table with rules defined by a certain flow mask. The flow masks are non-overlapping, and for flows matching more than one rule only the highest priority one is inserted in the corresponding sub-table (interested readers can refer to the Open vSwitch (OvS) design of Mega Flow Cache (MFC) [Member-OvS] for further details). Typically the rules will have a large number of distinct unique masks and hence, a large number of target sets each corresponding to one mask. Because the active set of flows varies widely based on the network traffic, HTSS with false negative will act as a cache for <flowid, target ACL sub-table> pair for the current active set of flows. When a miss occurs (as shown in red in the above figure) the sub-tables will be searched sequentially one by one for a possible match, and when found the flow key and target sub-table will be inserted into the set-summary (i.e. cache insertion) so subsequent packets from the same flow don't incur the overhead of the sequential search of sub-tables.

5.25.4 Library API Overview

The design goal of the Membership Library API is to be as generic as possible to support all the different types of set-summaries we discussed in previous sections and beyond. Fundamentally, the APIs need to include creation, insertion, deletion, and lookup.

Set-summary Create

The `rte_member_create()` function is used to create a set-summary structure, the input parameter is a struct to pass in parameters that needed to initialize the set-summary, while the function returns the pointer to the created set-summary or `NULL` if the creation failed.

The general input arguments used when creating the set-summary should include `name` which is the name of the created set-summary, *type* which is one of the types supported by the library (e.g. `RTE_MEMBER_TYPE_HT` for HTSS or `RTE_MEMBER_TYPE_VBF` for vBF), and `key_len` which is the length of the element/key. There are other parameters are only used for certain type of set-summary, or which have a slightly different meaning for different types of set-summary. For example, `num_keys` parameter means the maximum number of entries for Hash table based set-summary. However, for bloom filter, this value means the expected number of keys that could be inserted into the bloom filter(s). The value is used to calculate the size of each bloom filter.

We also pass two seeds: `prim_hash_seed` and `sec_hash_seed` for the primary and secondary hash functions to calculate two independent hash values. `socket_id` parameter is the NUMA socket ID for the memory used to create the set-summary. For HTSS, another parameter `is_cache` is used to indicate if this set-summary is a cache (i.e. with false negative probability) or not. For vBF, extra parameters are needed. For example, `num_set` is the number of sets needed to initialize the vector bloom filters. This number is equal to the number of bloom filters will be created. `false_pos_rate` is the false positive rate. num_keys and false_pos_rate will be used to determine the number of hash functions and the bloom filter size.

Set-summary Element Insertion

The `rte_member_add()` function is used to insert an element/key into a set-summary structure. If it fails an error is returned. For success the returned value is dependent on the set-summary mode to provide extra information for the users. For vBF mode, a return value of 0 means a successful insert. For HTSS mode without false negative, the insert could fail with `-ENOSPC` if the table is full. With false negative (i.e. cache mode), for insert that does not cause any eviction (i.e. no overwriting happens to an existing entry) the return value is 0. For insertion that causes eviction, the return value is 1 to indicate such situation, but it is not an error.

The input arguments for the function should include the `key` which is a pointer to the element/key that needs to be added to the set-summary, and `set_id` which is the set id associated with the key that needs to be added.

Set-summary Element Lookup

The `rte_member_lookup()` function looks up a single key/element in the set-summary structure. It returns as soon as the first match is found. The return value is 1 if a match is found and 0 otherwise. The arguments for the function include `key` which is a pointer to the element/key that needs to be looked up, and `set_id` which is used to return the first target set id where the key has matched, if any.

The `rte_member_lookup_bulk()` function is used to look up a bulk of keys/elements in the set-summary structure for their first match. Each key lookup returns as soon as the first match is found. The return value is the number of keys that find a match. The arguments of the function include `keys` which is a pointer to a bulk of keys that are to be looked up, `num_keys` is the number of keys that will be looked up, and `set_ids` are the return target set ids for the first match found for each of the input keys. `set_ids` is an array needs to be sized according to the `num_keys`. If there is no match, the set id for that key will be set to RTE_MEMBER_NO_MATCH.

The `rte_member_lookup_multi()` function looks up a single key/element in the set-summary structure for multiple matches. It returns ALL the matches (possibly more than one) found for this key when it is matched against all target sets (it is worth noting that for cache mode HTSS, the current implementation matches at most one target set). The return value is the number of matches that was found for this key (for cache mode HTSS the return value should be at most 1). The arguments for the function include `key` which is a pointer to the element/key that needs to be looked up, `max_match_per_key` which is to indicate the maximum number of matches the user expects to find for each key, and `set_id` which is used to return all target set ids where the key has matched, if any. The `set_id` array should be sized according to `max_match_per_key`. For vBF, the maximum number of matches per key is equal to the number of sets. For HTSS, the maximum number of matches per key is equal to two time entry count per bucket. `max_match_per_key` should be equal or smaller than the maximum number of possible matches.

The `rte_membership_lookup_multi_bulk()` function looks up a bulk of keys/elements in the set-summary structure for multiple matches, each key lookup returns ALL the matches (possibly more than one) found for this key when it is matched against all target sets (cache mode HTSS matches at most one target set). The return value is the number of keys that find one or more matches in the set-summary structure. The arguments of the function include `keys` which is a pointer to a bulk of keys that are to be looked up, `num_keys` is the number of keys that will be looked up, `max_match_per_key` is the possible maximum number of matches for each key, `match_count` which is the returned number of matches for each key, and `set_ids` are the returned target set ids for all matches found for each keys. `set_ids` is 2-D array containing a 1-D array for each key (the size of 1-D array per key should be set by the user according to `max_match_per_key`). `max_match_per_key` should be equal or smaller than the maximum number of possible matches, similar to `rte_member_lookup_multi`.

Set-summary Element Delete

The `rte_membership_delete()` function deletes an element/key from a set-summary structure, if it fails an error is returned. The input arguments should include `key` which is a pointer to the element/key that needs to be deleted from the set-summary, and `set_id` which is the set id associated with the key to delete. It is worth noting that current implementation of vBF does not support deletion[1]. An error code `-EINVAL` will be returned.

[1] Traditional bloom filter does not support proactive deletion. Supporting proactive deletion require additional implementation and performance overhead.

5.25.5 References

[Member-bloom] B H Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," Communications of the ACM, 1970.

[Member-survey] A Broder and M Mitzenmacher, "Network Applications of Bloom Filters: A Survey," in Internet Mathematics, 2005.

[Member-cfilter] B Fan, D G Andersen and M Kaminsky, "Cuckoo Filter: Practically Better Than Bloom," in Conference on emerging Networking Experiments and Technologies, 2014.

[Member-OvS] B Pfaff, "The Design and Implementation of Open vSwitch," in NSDI, 2015.

5.26 LPM Library

The DPDK LPM library component implements the Longest Prefix Match (LPM) table search method for 32-bit keys that is typically used to find the best route match in IP forwarding applications.

5.26.1 LPM API Overview

The main configuration parameter for LPM component instances is the maximum number of rules to support. An LPM prefix is represented by a pair of parameters (32- bit key, depth), with depth in the range of 1 to 32. An LPM rule is represented by an LPM prefix and some user data associated with the prefix. The prefix serves as the unique identifier of the LPM rule. In this implementation, the user data is 1-byte long and is called next hop, in correlation with its main use of storing the ID of the next hop in a routing table entry.

The main methods exported by the LPM component are:

- Add LPM rule: The LPM rule is provided as input. If there is no rule with the same prefix present in the table, then the new rule is added to the LPM table. If a rule with the same prefix is already present in the table, the next hop of the rule is updated. An error is returned when there is no available rule space left.
- Delete LPM rule: The prefix of the LPM rule is provided as input. If a rule with the specified prefix is present in the LPM table, then it is removed.
- Lookup LPM key: The 32-bit key is provided as input. The algorithm selects the rule that represents the best match for the given key and returns the next hop of that rule. In the case that there are multiple rules present in the LPM table that have the same 32-bit key, the algorithm picks the rule with the highest depth as the best match rule, which means that the rule has the highest number of most significant bits matching between the input key and the rule key.

5.26.2 Implementation Details

The current implementation uses a variation of the DIR-24-8 algorithm that trades memory usage for improved LPM lookup speed. The algorithm allows the lookup operation to be performed with typically a single memory read access. In the statistically rare case when the best match rule is having a depth bigger than 24, the lookup operation requires two memory read accesses. Therefore, the performance of the LPM lookup operation is greatly influenced by whether the specific memory location is present in the processor cache or not.

The main data structure is built using the following elements:

- A table with 2^24 entries.
- A number of tables (RTE_LPM_TBL8_NUM_GROUPS) with 2^8 entries.

The first table, called tbl24, is indexed using the first 24 bits of the IP address to be looked up, while the second table(s), called tbl8, is indexed using the last 8 bits of the IP address. This means that depending on the outcome of trying to match the IP address of an incoming packet to the rule stored in the tbl24 we might need to continue the lookup process in the second level.

Since every entry of the tbl24 can potentially point to a tbl8, ideally, we would have 2^24 tbl8s, which would be the same as having a single table with 2^32 entries. This is not feasible due to resource restrictions. Instead, this approach takes advantage of the fact that rules longer than 24 bits are very rare. By splitting the process in two different tables/levels and limiting the number of tbl8s, we can greatly reduce memory consumption while maintaining a very good lookup speed (one memory access, most of the times).

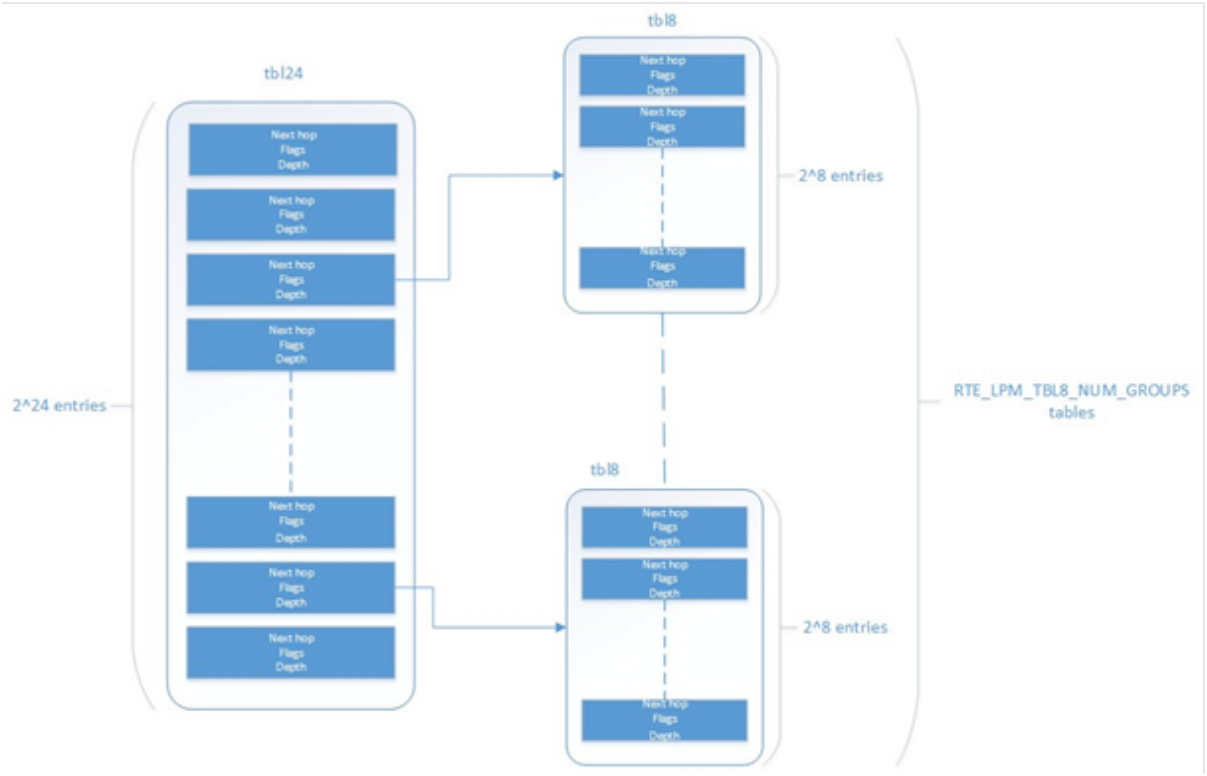


Fig. 5.55: Table split into different levels

An entry in tbl24 contains the following fields:

- next hop / index to the tbl8

- valid flag
- external entry flag
- depth of the rule (length)

The first field can either contain a number indicating the tbl8 in which the lookup process should continue or the next hop itself if the longest prefix match has already been found. The two flags are used to determine whether the entry is valid or not and whether the search process have finished or not respectively. The depth or length of the rule is the number of bits of the rule that is stored in a specific entry.

An entry in a tbl8 contains the following fields:

- next hop
- valid
- valid group
- depth

Next hop and depth contain the same information as in the tbl24. The two flags show whether the entry and the table are valid respectively.

The other main data structure is a table containing the main information about the rules (IP and next hop). This is a higher level table, used for different things:

- Check whether a rule already exists or not, prior to addition or deletion, without having to actually perform a lookup.
- When deleting, to check whether there is a rule containing the one that is to be deleted. This is important, since the main data structure will have to be updated accordingly.

Addition

When adding a rule, there are different possibilities. If the rule's depth is exactly 24 bits, then:

- Use the rule (IP address) as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then set its next hop to its value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 0 (meaning the lookup process ends at this point, since this is the longest prefix that matches).

If the rule's depth is exactly 32 bits, then:

- Use the first 24 bits of the rule as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then look for a free tbl8, set the index to the tbl8 to this value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 1 (meaning the lookup process must continue since the rule hasn't been explored completely).

If the rule's depth is any other value, prefix expansion must be performed. This means the rule is copied to all the entries (as long as they are not in use) which would also cause a match.

As a simple example, let's assume the depth is 20 bits. This means that there are 2^(24 - 20) = 16 different combinations of the first 24 bits of an IP address that would cause a match. Hence, in this case, we copy the exact same entry to every position indexed by one of these combinations.

By doing this we ensure that during the lookup process, if a rule matching the IP address exists, it is found in either one or two memory accesses, depending on whether we need to move to the next table

or not. Prefix expansion is one of the keys of this algorithm, since it improves the speed dramatically by adding redundancy.

Lookup

The lookup process is much simpler and quicker. In this case:

- Use the first 24 bits of the IP address as an index to the tbl24. If the entry is not in use, then it means we don't have a rule matching this IP. If it is valid and the external entry flag is set to 0, then the next hop is returned.
- If it is valid and the external entry flag is set to 1, then we use the tbl8 index to find out the tbl8 to be checked, and the last 8 bits of the IP address as an index to this table. Similarly, if the entry is not in use, then we don't have a rule matching this IP address. If it is valid then the next hop is returned.

Limitations in the Number of Rules

There are different things that limit the number of rules that can be added. The first one is the maximum number of rules, which is a parameter passed through the API. Once this number is reached, it is not possible to add any more rules to the routing table unless one or more are removed.

The second reason is an intrinsic limitation of the algorithm. As explained before, to avoid high memory consumption, the number of tbl8s is limited in compilation time (this value is by default 256). If we exhaust tbl8s, we won't be able to add any more rules. How many of them are necessary for a specific routing table is hard to determine in advance.

A tbl8 is consumed whenever we have a new rule with depth bigger than 24, and the first 24 bits of this rule are not the same as the first 24 bits of a rule previously added. If they are, then the new rule will share the same tbl8 than the previous one, since the only difference between the two rules is within the last byte.

With the default value of 256, we can have up to 256 rules longer than 24 bits that differ on their first three bytes. Since routes longer than 24 bits are unlikely, this shouldn't be a problem in most setups. Even if it is, however, the number of tbl8s can be modified.

Use Case: IPv4 Forwarding

The LPM algorithm is used to implement Classless Inter-Domain Routing (CIDR) strategy used by routers implementing IPv4 forwarding.

References

- RFC1519 Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy, http://www.ietf.org/rfc/rfc1519
- Pankaj Gupta, Algorithms for Routing Lookups and Packet Classification, PhD Thesis, Stanford University, 2000 (http://klamath.stanford.edu/~pankaj/thesis/thesis_1sided.pdf)

5.27 LPM6 Library

The LPM6 (LPM for IPv6) library component implements the Longest Prefix Match (LPM) table search method for 128-bit keys that is typically used to find the best match route in IPv6 forwarding applications.

5.27.1 LPM6 API Overview

The main configuration parameters for the LPM6 library are:

- Maximum number of rules: This defines the size of the table that holds the rules, and therefore the maximum number of rules that can be added.
- Number of tbl8s: A tbl8 is a node of the trie that the LPM6 algorithm is based on.

This parameter is related to the number of rules you can have, but there is no way to accurately predict the number needed to hold a specific number of rules, since it strongly depends on the depth and IP address of every rule. One tbl8 consumes 1 kb of memory. As a recommendation, 65536 tbl8s should be sufficient to store several thousand IPv6 rules, but the number can vary depending on the case.

An LPM prefix is represented by a pair of parameters (128-bit key, depth), with depth in the range of 1 to 128. An LPM rule is represented by an LPM prefix and some user data associated with the prefix. The prefix serves as the unique identifier for the LPM rule. In this implementation, the user data is 21-bits long and is called "next hop", which corresponds to its main use of storing the ID of the next hop in a routing table entry.

The main methods exported for the LPM component are:

- Add LPM rule: The LPM rule is provided as input. If there is no rule with the same prefix present in the table, then the new rule is added to the LPM table. If a rule with the same prefix is already present in the table, the next hop of the rule is updated. An error is returned when there is no available space left.
- Delete LPM rule: The prefix of the LPM rule is provided as input. If a rule with the specified prefix is present in the LPM table, then it is removed.
- Lookup LPM key: The 128-bit key is provided as input. The algorithm selects the rule that represents the best match for the given key and returns the next hop of that rule. In the case that there are multiple rules present in the LPM table that have the same 128-bit value, the algorithm picks the rule with the highest depth as the best match rule, which means the rule has the highest number of most significant bits matching between the input key and the rule key.

Implementation Details

This is a modification of the algorithm used for IPv4 (see *Implementation Details*). In this case, instead of using two levels, one with a tbl24 and a second with a tbl8, 14 levels are used.

The implementation can be seen as a multi-bit trie where the *stride* or number of bits inspected on each level varies from level to level. Specifically, 24 bits are inspected on the root node, and the remaining 104 bits are inspected in groups of 8 bits. This effectively means that the trie has 14 levels at the most, depending on the rules that are added to the table.

The algorithm allows the lookup operation to be performed with a number of memory accesses that directly depends on the length of the rule and whether there are other rules with bigger depths and the same key in the data structure. It can vary from 1 to 14 memory accesses, with 5 being the average value for the lengths that are most commonly used in IPv6.

The main data structure is built using the following elements:

- A table with 224 entries
- A number of tables, configurable by the user through the API, with 28 entries

The first table, called tbl24, is indexed using the first 24 bits of the IP address be looked up, while the rest of the tables, called tbl8s, are indexed using the rest of the bytes of the IP address, in chunks of 8 bits. This means that depending on the outcome of trying to match the IP address of an incoming packet to the rule stored in the tbl24 or the subsequent tbl8s we might need to continue the lookup process in deeper levels of the tree.

Similar to the limitation presented in the algorithm for IPv4, to store every possible IPv6 rule, we would need a table with 2^128 entries. This is not feasible due to resource restrictions.

By splitting the process in different tables/levels and limiting the number of tbl8s, we can greatly reduce memory consumption while maintaining a very good lookup speed (one memory access per level).

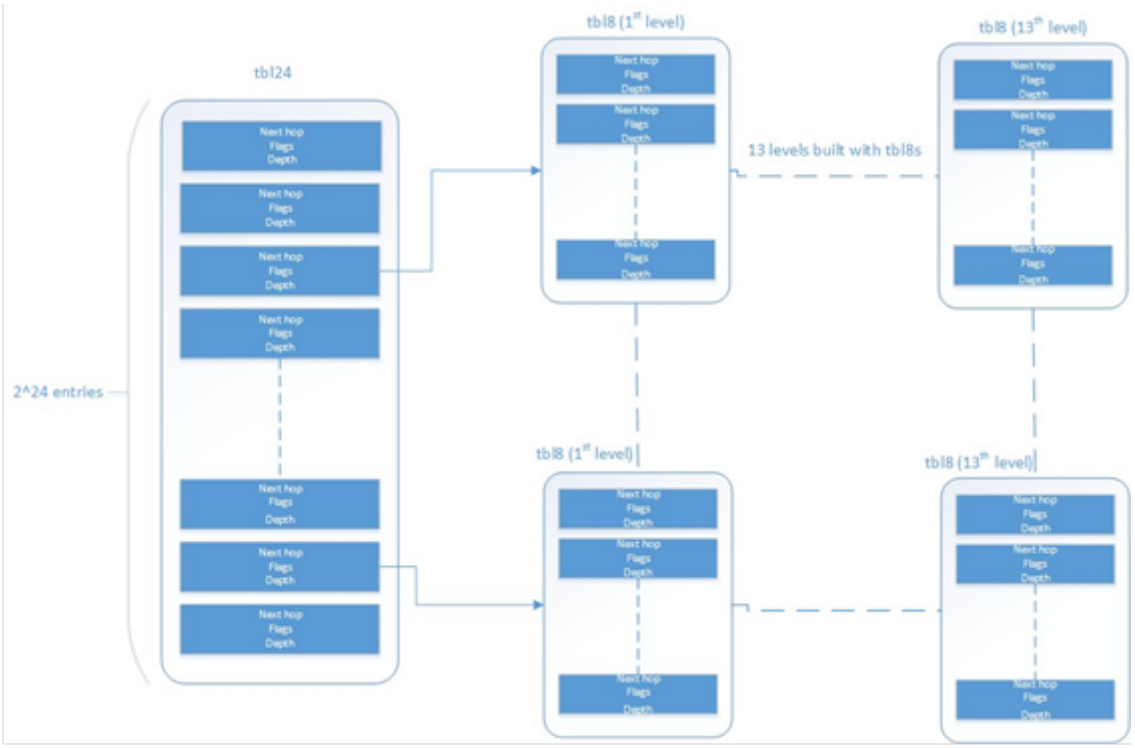


Fig. 5.56: Table split into different levels

An entry in a table contains the following fields:

- next hop / index to the tbl8
- depth of the rule (length)
- valid flag
- valid group flag
- external entry flag

The first field can either contain a number indicating the tbl8 in which the lookup process should continue or the next hop itself if the longest prefix match has already been found. The depth or length of the rule is the number of bits of the rule that is stored in a specific entry. The flags are used to determine whether the entry/table is valid or not and whether the search process have finished or not respectively.

Both types of tables share the same structure.

The other main data structure is a table containing the main information about the rules (IP, next hop and depth). This is a higher level table, used for different things:

- Check whether a rule already exists or not, prior to addition or deletion, without having to actually perform a lookup.

When deleting, to check whether there is a rule containing the one that is to be deleted. This is important, since the main data structure will have to be updated accordingly.

Addition

When adding a rule, there are different possibilities. If the rule's depth is exactly 24 bits, then:

- Use the rule (IP address) as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then set its next hop to its value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 0 (meaning the lookup process ends at this point, since this is the longest prefix that matches).

If the rule's depth is bigger than 24 bits but a multiple of 8, then:

- Use the first 24 bits of the rule as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then look for a free tbl8, set the index to the tbl8 to this value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 1 (meaning the lookup process must continue since the rule hasn't been explored completely).
- Use the following 8 bits of the rule as an index to the next tbl8.
- Repeat the process until the tbl8 at the right level (depending on the depth) has been reached and fill it with the next hop, setting the next entry flag to 0.

If the rule's depth is any other value, prefix expansion must be performed. This means the rule is copied to all the entries (as long as they are not in use) which would also cause a match.

As a simple example, let's assume the depth is 20 bits. This means that there are 2^(24-20) = 16 different combinations of the first 24 bits of an IP address that would cause a match. Hence, in this case, we copy the exact same entry to every position indexed by one of these combinations.

By doing this we ensure that during the lookup process, if a rule matching the IP address exists, it is found in, at the most, 14 memory accesses, depending on how many times we need to move to the next table. Prefix expansion is one of the keys of this algorithm, since it improves the speed dramatically by adding redundancy.

Prefix expansion can be performed at any level. So, for example, is the depth is 34 bits, it will be performed in the third level (second tbl8-based level).

Lookup

The lookup process is much simpler and quicker. In this case:

- Use the first 24 bits of the IP address as an index to the tbl24. If the entry is not in use, then it means we don't have a rule matching this IP. If it is valid and the external entry flag is set to 0, then the next hop is returned.
- If it is valid and the external entry flag is set to 1, then we use the tbl8 index to find out the tbl8 to be checked, and the next 8 bits of the IP address as an index to this table. Similarly, if the entry is not in use, then we don't have a rule matching this IP address. If it is valid then check the external entry flag for a new tbl8 to be inspected.
- Repeat the process until either we find an invalid entry (lookup miss) or a valid entry with the external entry flag set to 0. Return the next hop in the latter case.

Limitations in the Number of Rules

There are different things that limit the number of rules that can be added. The first one is the maximum number of rules, which is a parameter passed through the API. Once this number is reached, it is not possible to add any more rules to the routing table unless one or more are removed.

The second limitation is in the number of tbl8s available. If we exhaust tbl8s, we won't be able to add any more rules. How to know how many of them are necessary for a specific routing table is hard to determine in advance.

In this algorithm, the maximum number of tbl8s a single rule can consume is 13, which is the number of levels minus one, since the first three bytes are resolved in the tbl24. However:

- Typically, on IPv6, routes are not longer than 48 bits, which means rules usually take up to 3 tbl8s.

As explained in the LPM for IPv4 algorithm, it is possible and very likely that several rules will share one or more tbl8s, depending on what their first bytes are. If they share the same first 24 bits, for instance, the tbl8 at the second level will be shared. This might happen again in deeper levels, so, effectively, two 48 bit-long rules may use the same three tbl8s if the only difference is in their last byte.

The number of tbl8s is a parameter exposed to the user through the API in this version of the algorithm, due to its impact in memory consumption and the number or rules that can be added to the LPM table. One tbl8 consumes 1 kilobyte of memory.

5.27.2 Use Case: IPv6 Forwarding

The LPM algorithm is used to implement the Classless Inter-Domain Routing (CIDR) strategy used by routers implementing IP forwarding.

5.28 Flow Classification Library

DPDK provides a Flow Classification library that provides the ability to classify an input packet by matching it against a set of Flow rules.

The initial implementation supports counting of IPv4 5-tuple packets which match a particular Flow rule only.

Please refer to the *Generic flow API (rte_flow)* for more information.

The Flow Classification library uses the `librte_table` API for managing Flow rules and matching packets against the Flow rules. The library is table agnostic and can use the following tables: `Access Control List`, `Hash` and `Longest Prefix Match(LPM)`. The `Access Control List` table is used in the initial implementation.

Please refer to the *Packet Framework* for more information.on `librte_table`.

DPDK provides an Access Control List library that provides the ability to classify an input packet based on a set of classification rules.

Please refer to the *Packet Classification and Access Control* library for more information on `librte_acl`.

There is also a Flow Classify sample application which demonstrates the use of the Flow Classification Library API's.

Please refer to the *Flow Classify Sample Application* for more information on the `flow_classify` sample application.

5.28.1 Overview

The library has the following API's

```
/**
 * Flow classifier create
 *
 * @param params
 *   Parameters for flow classifier creation
 * @return
 *   Handle to flow classifier instance on success or NULL otherwise
 */
struct rte_flow_classifier *
rte_flow_classifier_create(struct rte_flow_classifier_params *params);

/**
 * Flow classifier free
 *
 * @param cls
 *   Handle to flow classifier instance
 * @return
 *   0 on success, error code otherwise
 */
int
rte_flow_classifier_free(struct rte_flow_classifier *cls);

/**
 * Flow classify table create
 *
 * @param cls
 *   Handle to flow classifier instance
```

(continues on next page)

(continued from previous page)

```
 * @param params
 *   Parameters for flow_classify table creation
 * @return
 *   0 on success, error code otherwise
 */
int
rte_flow_classify_table_create(struct rte_flow_classifier *cls,
       struct rte_flow_classify_table_params *params);

/**
 * Validate the flow classify rule
 *
 * @param[in] cls
 *   Handle to flow classifier instance
 * @param[in] attr
 *   Flow rule attributes
 * @param[in] pattern
 *   Pattern specification (list terminated by the END pattern item).
 * @param[in] actions
 *   Associated actions (list terminated by the END pattern item).
 * @param[out] error
 *   Perform verbose error reporting if not NULL. Structure
 *   initialised in case of error only.
 * @return
 *   0 on success, error code otherwise
 */
int
rte_flow_classify_validate(struct rte_flow_classifier *cls,
        const struct rte_flow_attr *attr,
        const struct rte_flow_item pattern[],
        const struct rte_flow_action actions[],
        struct rte_flow_error *error);

/**
 * Add a flow classify rule to the flow_classifier table.
 *
 * @param[in] cls
 *   Flow classifier handle
 * @param[in] attr
 *   Flow rule attributes
 * @param[in] pattern
 *   Pattern specification (list terminated by the END pattern item).
 * @param[in] actions
 *   Associated actions (list terminated by the END pattern item).
 * @param[out] key_found
 *   returns 1 if rule present already, 0 otherwise.
 * @param[out] error
 *   Perform verbose error reporting if not NULL. Structure
 *   initialised in case of error only.
 * @return
 *   A valid handle in case of success, NULL otherwise.
 */
struct rte_flow_classify_rule *
rte_flow_classify_table_entry_add(struct rte_flow_classifier *cls,
        const struct rte_flow_attr *attr,
        const struct rte_flow_item pattern[],
        const struct rte_flow_action actions[],
        int *key_found;
        struct rte_flow_error *error);

/**
```

(continues on next page)

(continued from previous page)

```
 * Delete a flow classify rule from the flow_classifier table.
 *
 * @param[in] cls
 *   Flow classifier handle
 * @param[in] rule
 *   Flow classify rule
 * @return
 *   0 on success, error code otherwise.
 */
int
rte_flow_classify_table_entry_delete(struct rte_flow_classifier *cls,
        struct rte_flow_classify_rule *rule);

/**
 * Query flow classifier for given rule.
 *
 * @param[in] cls
 *   Flow classifier handle
 * @param[in] pkts
 *   Pointer to packets to process
 * @param[in] nb_pkts
 *   Number of packets to process
 * @param[in] rule
 *   Flow classify rule
 * @param[in] stats
 *   Flow classify stats
 *
 * @return
 *   0 on success, error code otherwise.
 */
int
rte_flow_classifier_query(struct rte_flow_classifier *cls,
        struct rte_mbuf **pkts,
        const uint16_t nb_pkts,
        struct rte_flow_classify_rule *rule,
        struct rte_flow_classify_stats *stats);
```

Classifier creation

The application creates the `Classifier` using the `rte_flow_classifier_create` API. The `rte_flow_classify_params` structure must be initialised by the application before calling the API.

```
struct rte_flow_classifier_params {
    /** flow classifier name */
    const char *name;

    /** CPU socket ID where memory for the flow classifier and its */
    /** elements (tables) should be allocated */
    int socket_id;
};
```

The `Classifier` has the following internal structures:

```
struct rte_cls_table {
    /* Input parameters */
    struct rte_table_ops ops;
    uint32_t entry_size;
    enum rte_flow_classify_table_type type;
```

(continues on next page)

(continued from previous page)

```
    /* Handle to the low-level table object */
    void *h_table;
};

#define RTE_FLOW_CLASSIFIER_MAX_NAME_SZ 256

struct rte_flow_classifier {
    /* Input parameters */
    char name[RTE_FLOW_CLASSIFIER_MAX_NAME_SZ];
    int socket_id;

    /* Internal */
    /* ntuple_filter */
    struct rte_eth_ntuple_filter ntuple_filter;

    /* classifier tables */
    struct rte_cls_table tables[RTE_FLOW_CLASSIFY_TABLE_MAX];
    uint32_t table_mask;
    uint32_t num_tables;

    uint16_t nb_pkts;
    struct rte_flow_classify_table_entry
        *entries[RTE_PORT_IN_BURST_SIZE_MAX];
} __rte_cache_aligned;
```

Adding a table to the Classifier

The application adds a table to the `Classifier` using the `rte_flow_classify_table_create` API. The `rte_flow_classify_table_params` structure must be initialised by the application before calling the API.

```
struct rte_flow_classify_table_params {
    /** Table operations (specific to each table type) */
    struct rte_table_ops *ops;

    /** Opaque param to be passed to the table create operation */
    void *arg_create;

    /** Classifier table type */
    enum rte_flow_classify_table_type type;
 };
```

To create an ACL table the `rte_table_acl_params` structure must be initialised and assigned to `arg_create` in the `rte_flow_classify_table_params` structure.

```
struct rte_table_acl_params {
    /** Name */
    const char *name;

    /** Maximum number of ACL rules in the table */
    uint32_t n_rules;

    /** Number of fields in the ACL rule specification */
    uint32_t n_rule_fields;

    /** Format specification of the fields of the ACL rule */
```

(continues on next page)

(continued from previous page)

```
    struct rte_acl_field_def field_format[RTE_ACL_MAX_FIELDS];
};
```

The fields for the ACL rule must also be initialised by the application.

An ACL table can be added to the `Classifier` for each ACL rule, for example another table could be added for the IPv6 5-tuple rule.

Flow Parsing

The library currently supports three IPv4 5-tuple flow patterns, for UDP, TCP and SCTP.

```
/* Pattern for IPv4 5-tuple UDP filter */
static enum rte_flow_item_type pattern_ntuple_1[] = {
    RTE_FLOW_ITEM_TYPE_ETH,
    RTE_FLOW_ITEM_TYPE_IPV4,
    RTE_FLOW_ITEM_TYPE_UDP,
    RTE_FLOW_ITEM_TYPE_END,
};

/* Pattern for IPv4 5-tuple TCP filter */
static enum rte_flow_item_type pattern_ntuple_2[] = {
    RTE_FLOW_ITEM_TYPE_ETH,
    RTE_FLOW_ITEM_TYPE_IPV4,
    RTE_FLOW_ITEM_TYPE_TCP,
    RTE_FLOW_ITEM_TYPE_END,
};

/* Pattern for IPv4 5-tuple SCTP filter */
static enum rte_flow_item_type pattern_ntuple_3[] = {
    RTE_FLOW_ITEM_TYPE_ETH,
    RTE_FLOW_ITEM_TYPE_IPV4,
    RTE_FLOW_ITEM_TYPE_SCTP,
    RTE_FLOW_ITEM_TYPE_END,
};
```

The API function `rte_flow_classify_validate` parses the IPv4 5-tuple pattern, attributes and actions and returns the 5-tuple data in the `rte_eth_ntuple_filter` structure.

```
static int
rte_flow_classify_validate(struct rte_flow_classifier *cls,
                const struct rte_flow_attr *attr,
                const struct rte_flow_item pattern[],
                const struct rte_flow_action actions[],
                struct rte_flow_error *error)
```

Adding Flow Rules

The `rte_flow_classify_table_entry_add` API creates an `rte_flow_classify` object which contains the flow_classify id and type, the action, a union of add and delete keys and a union of rules. It uses the `rte_flow_classify_validate` API function for parsing the flow parameters. The 5-tuple ACL key data is obtained from the `rte_eth_ntuple_filter` structure populated by the `classify_parse_ntuple_filter` function which parses the Flow rule.

```
struct acl_keys {
    struct rte_table_acl_rule_add_params key_add; /* add key */
    struct rte_table_acl_rule_delete_params key_del; /* delete key */
};

struct classify_rules {
    enum rte_flow_classify_rule_type type;
    union {
        struct rte_flow_classify_ipv4_5tuple ipv4_5tuple;
    } u;
};

struct rte_flow_classify {
    uint32_t id;  /* unique ID of classify object */
    enum rte_flow_classify_table_type tbl_type; /* rule table */
    struct classify_rules rules; /* union of rules */
    union {
        struct acl_keys key;
    } u;
    int key_found; /* rule key found in table */
    struct rte_flow_classify_table_entry entry;  /* rule meta data */
    void *entry_ptr; /* handle to the table entry for rule meta data */
};
```

It then calls the `table.ops.f_add` API to add the rule to the ACL table.

Deleting Flow Rules

The `rte_flow_classify_table_entry_delete` API calls the `table.ops.f_delete` API to delete a rule from the ACL table.

Packet Matching

The `rte_flow_classifier_query` API is used to find packets which match a given flow Flow rule in the table. This API calls the flow_classify_run internal function which calls the `table.ops.f_lookup` API to see if any packets in a burst match any of the Flow rules in the table. The meta data for the highest priority rule matched for each packet is returned in the entries array in the `rte_flow_classify` object. The internal function `action_apply` implements the `Count` action which is used to return data which matches a particular Flow rule.

The rte_flow_classifier_query API uses the following structures to return data to the application.

```
/** IPv4 5-tuple data */
struct rte_flow_classify_ipv4_5tuple {
    uint32_t dst_ip;         /**< Destination IP address in big endian. */
    uint32_t dst_ip_mask;    /**< Mask of destination IP address. */
    uint32_t src_ip;         /**< Source IP address in big endian. */
    uint32_t src_ip_mask;    /**< Mask of destination IP address. */
    uint16_t dst_port;       /**< Destination port in big endian. */
    uint16_t dst_port_mask;  /**< Mask of destination port. */
    uint16_t src_port;       /**< Source Port in big endian. */
    uint16_t src_port_mask;  /**< Mask of source port. */
    uint8_t proto;           /**< L4 protocol. */
    uint8_t proto_mask;      /**< Mask of L4 protocol. */
};

/**
```

(continues on next page)

(continued from previous page)

```
 * Flow stats
 *
 * For the count action, stats can be returned by the query API.
 *
 * Storage for stats is provided by the application.
 *
 *
 */
struct rte_flow_classify_stats {
    void *stats;
};

struct rte_flow_classify_5tuple_stats {
    /** count of packets that match IPv4 5tuple pattern */
    uint64_t counter1;
    /** IPv4 5tuple data */
    struct rte_flow_classify_ipv4_5tuple ipv4_5tuple;
};
```

5.29 Packet Distributor Library

The DPDK Packet Distributor library is a library designed to be used for dynamic load balancing of traffic while supporting single packet at a time operation. When using this library, the logical cores in use are to be considered in two roles: firstly a distributor lcore, which is responsible for load balancing or distributing packets, and a set of worker lcores which are responsible for receiving the packets from the distributor and operating on them. The model of operation is shown in the diagram below.

There are two modes of operation of the API in the distributor library, one which sends one packet at a time to workers using 32-bits for flow_id, and an optimized mode which sends bursts of up to 8 packets at a time to workers, using 15 bits of flow_id. The mode is selected by the type field in the `rte_distributor_create()` function.

5.29.1 Distributor Core Operation

The distributor core does the majority of the processing for ensuring that packets are fairly shared among workers. The operation of the distributor is as follows:

1. Packets are passed to the distributor component by having the distributor lcore thread call the "rte_distributor_process()" API
2. The worker lcores all share a single cache line with the distributor core in order to pass messages and packets to and from the worker. The process API call will poll all the worker cache lines to see what workers are requesting packets.
3. As workers request packets, the distributor takes packets from the set of packets passed in and distributes them to the workers. As it does so, it examines the "tag" – stored in the RSS hash field in the mbuf – for each packet and records what tags are being processed by each worker.
4. If the next packet in the input set has a tag which is already being processed by a worker, then that packet will be queued up for processing by that worker and given to it in preference to other packets when that work next makes a request for work. This ensures that no two packets with the same tag are processed in parallel, and that all packets with the same tag are processed in input order.

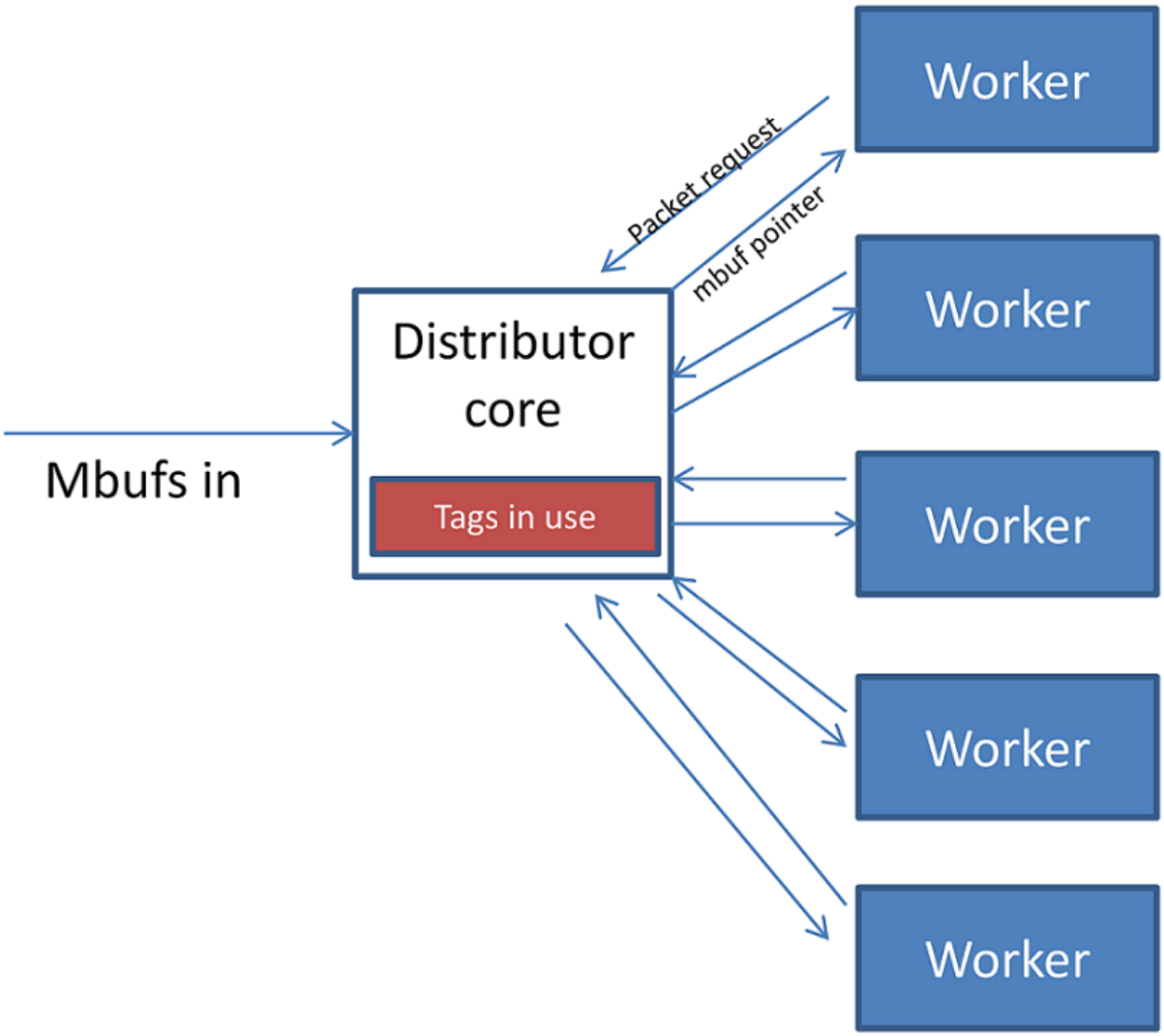


Fig. 5.57: Packet Distributor mode of operation

5. Once all input packets passed to the process API have either been distributed to workers or been queued up for a worker which is processing a given tag, then the process API returns to the caller.

Other functions which are available to the distributor lcore are:

- rte_distributor_returned_pkts()
- rte_distributor_flush()
- rte_distributor_clear_returns()

Of these the most important API call is "rte_distributor_returned_pkts()" which should only be called on the lcore which also calls the process API. It returns to the caller all packets which have finished processing by all worker cores. Within this set of returned packets, all packets sharing the same tag will be returned in their original order.

NOTE: If worker lcores buffer up packets internally for transmission in bulk afterwards, the packets sharing a tag will likely get out of order. Once a worker lcore requests a new packet, the distributor assumes that it has completely finished with the previous packet and therefore that additional packets with the same tag can safely be distributed to other workers – who may then flush their buffered packets sooner and cause packets to get out of order.

NOTE: No packet ordering guarantees are made about packets which do not share a common packet tag.

Using the process and returned_pkts API, the following application workflow can be used, while allowing packet order within a packet flow – identified by a tag – to be maintained.

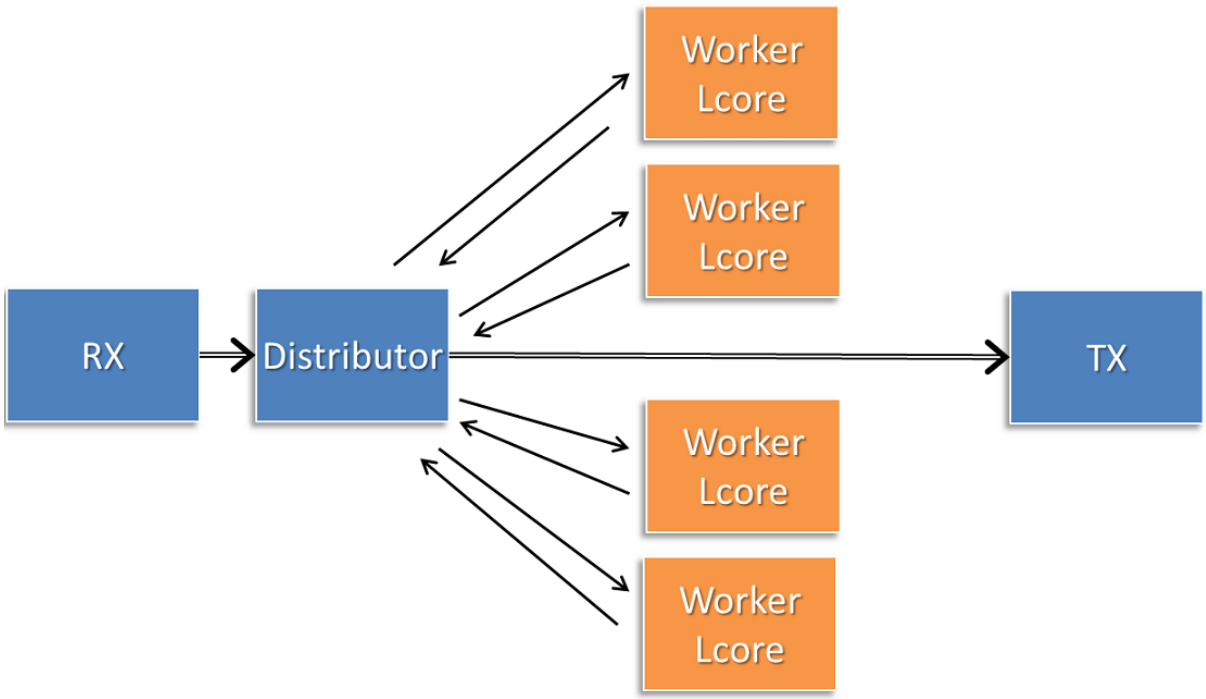


Fig. 5.58: Application workflow

The flush and clear_returns API calls, mentioned previously, are likely of less use that the process and returned_pkts APIS, and are principally provided to aid in unit testing of the library. Descriptions of these functions and their use can be found in the DPDK API Reference document.

5.29.2 Worker Operation

Worker cores are the cores which do the actual manipulation of the packets distributed by the packet distributor. Each worker calls "rte_distributor_get_pkt()" API to request a new packet when it has finished processing the previous one. [The previous packet should be returned to the distributor component by passing it as the final parameter to this API call.]

Since it may be desirable to vary the number of worker cores, depending on the traffic load i.e. to save power at times of lighter load, it is possible to have a worker stop processing packets by calling "rte_distributor_return_pkt()" to indicate that it has finished the current packet and does not want a new one.

5.30 Reorder Library

The Reorder Library provides a mechanism for reordering mbufs based on their sequence number.

5.30.1 Operation

The reorder library is essentially a buffer that reorders mbufs. The user inserts out of order mbufs into the reorder buffer and pulls in-order mbufs from it.

At a given time, the reorder buffer contains mbufs whose sequence number are inside the sequence window. The sequence window is determined by the minimum sequence number and the number of entries that the buffer was configured to hold. For example, given a reorder buffer with 200 entries and a minimum sequence number of 350, the sequence window has low and high limits of 350 and 550 respectively.

When inserting mbufs, the reorder library differentiates between valid, early and late mbufs depending on the sequence number of the inserted mbuf:

- valid: the sequence number is inside the window.
- late: the sequence number is outside the window and less than the low limit.
- early: the sequence number is outside the window and greater than the high limit.

The reorder buffer directly returns late mbufs and tries to accommodate early mbufs.

5.30.2 Implementation Details

The reorder library is implemented as a pair of buffers, which referred to as the *Order* buffer and the *Ready* buffer.

On an insert call, valid mbufs are inserted directly into the Order buffer and late mbufs are returned to the user with an error.

In the case of early mbufs, the reorder buffer will try to move the window (incrementing the minimum sequence number) so that the mbuf becomes a valid one. To that end, mbufs in the Order buffer are moved into the Ready buffer. Any mbufs that have not arrived yet are ignored and therefore will become late mbufs. This means that as long as there is room in the Ready buffer, the window will be moved to accommodate early mbufs that would otherwise be outside the reordering window.

For example, assuming that we have a buffer of 200 entries with a 350 minimum sequence number, and we need to insert an early mbuf with 565 sequence number. That means that we would need to move the windows at least 15 positions to accommodate the mbuf. The reorder buffer would try to move mbufs

from at least the next 15 slots in the Order buffer to the Ready buffer, as long as there is room in the Ready buffer. Any gaps in the Order buffer at that point are skipped, and those packet will be reported as late packets when they arrive. The process of moving packets to the Ready buffer continues beyond the minimum required until a gap, i.e. missing mbuf, in the Order buffer is encountered.

When draining mbufs, the reorder buffer would return mbufs in the Ready buffer first and then from the Order buffer until a gap is found (mbufs that have not arrived yet).

5.30.3 Use Case: Packet Distributor

An application using the DPDK packet distributor could make use of the reorder library to transmit packets in the same order they were received.

A basic packet distributor use case would consist of a distributor with multiple workers cores. The processing of packets by the workers is not guaranteed to be in order, hence a reorder buffer can be used to order as many packets as possible.

In such a scenario, the distributor assigns a sequence number to mbufs before delivering them to the workers. As the workers finish processing the packets, the distributor inserts those mbufs into the reorder buffer and finally transmit drained mbufs.

NOTE: Currently the reorder buffer is not thread safe so the same thread is responsible for inserting and draining mbufs.

5.31 IP Fragmentation and Reassembly Library

The IP Fragmentation and Reassembly Library implements IPv4 and IPv6 packet fragmentation and reassembly.

5.31.1 Packet fragmentation

Packet fragmentation routines divide input packet into number of fragments. Both rte_ipv4_fragment_packet() and rte_ipv6_fragment_packet() functions assume that input mbuf data points to the start of the IP header of the packet (i.e. L2 header is already stripped out). To avoid copying of the actual packet's data zero-copy technique is used (rte_pktmbuf_attach). For each fragment two new mbufs are created:

- Direct mbuf – mbuf that will contain L3 header of the new fragment.
- Indirect mbuf – mbuf that is attached to the mbuf with the original packet. It's data field points to the start of the original packets data plus fragment offset.

Then L3 header is copied from the original mbuf into the 'direct' mbuf and updated to reflect new fragmented status. Note that for IPv4, header checksum is not recalculated and is set to zero.

Finally 'direct' and 'indirect' mbufs for each fragment are linked together via mbuf's next filed to compose a packet for the new fragment.

The caller has an ability to explicitly specify which mempools should be used to allocate 'direct' and 'indirect' mbufs from.

For more information about direct and indirect mbufs, refer to *Direct and Indirect Buffers*.

5.31.2 Packet reassembly

IP Fragment Table

Fragment table maintains information about already received fragments of the packet.

Each IP packet is uniquely identified by triple <Source IP address>, <Destination IP address>, <ID>.

Note that all update/lookup operations on Fragment Table are not thread safe. So if different execution contexts (threads/processes) will access the same table simultaneously, then some external syncing mechanism have to be provided.

Each table entry can hold information about packets consisting of up to RTE_LIBRTE_IP_FRAG_MAX (by default: 4) fragments.

Code example, that demonstrates creation of a new Fragment table:

```
frag_cycles = (rte_get_tsc_hz() + MS_PER_S - 1) / MS_PER_S * max_flow_ttl;
bucket_num = max_flow_num + max_flow_num / 4;
frag_tbl = rte_ip_frag_table_create(max_flow_num, bucket_entries, max_flow_num, frag_cycles,␣
↪socket_id);
```

Internally Fragment table is a simple hash table. The basic idea is to use two hash functions and <bucket_entries> * associativity. This provides 2 * <bucket_entries> possible locations in the hash table for each key. When the collision occurs and all 2 * <bucket_entries> are occupied, instead of reinserting existing keys into alternative locations, ip_frag_tbl_add() just returns a failure.

Also, entries that resides in the table longer then <max_cycles> are considered as invalid, and could be removed/replaced by the new ones.

Note that reassembly demands a lot of mbuf's to be allocated. At any given time up to (2 * bucket_entries * RTE_LIBRTE_IP_FRAG_MAX * <maximum number of mbufs per packet>) can be stored inside Fragment Table waiting for remaining fragments.

Packet Reassembly

Fragmented packets processing and reassembly is done by the rte_ipv4_frag_reassemble_packet()/rte_ipv6_frag_reassemble_packet. Functions. They either return a pointer to valid mbuf that contains reassembled packet, or NULL (if the packet can't be reassembled for some reason).

These functions are responsible for:

1. Search the Fragment Table for entry with packet's <IPv4 Source Address, IPv4 Destination Address, Packet ID>.
2. If the entry is found, then check if that entry already timed-out. If yes, then free all previously received fragments, and remove information about them from the entry.
3. If no entry with such key is found, then try to create a new one by one of two ways:
   a) Use as empty entry.
   b) Delete a timed-out entry, free mbufs associated with it mbufs and store a new entry with specified key in it.
4. Update the entry with new fragment information and check if a packet can be reassembled (the packet's entry contains all fragments).

a) If yes, then, reassemble the packet, mark table's entry as empty and return the reassembled mbuf to the caller.

b) If no, then return a NULL to the caller.

If at any stage of packet processing an error is encountered (e.g: can't insert new entry into the Fragment Table, or invalid/timed-out fragment), then the function will free all associated with the packet fragments, mark the table entry as invalid and return NULL to the caller.

Debug logging and Statistics Collection

The RTE_LIBRTE_IP_FRAG_TBL_STAT config macro controls statistics collection for the Fragment Table. This macro is not enabled by default.

The RTE_LIBRTE_IP_FRAG_DEBUG controls debug logging of IP fragments processing and reassembling. This macro is disabled by default. Note that while logging contains a lot of detailed information, it slows down packet processing and might cause the loss of a lot of packets.

5.32 Generic Receive Offload Library

Generic Receive Offload (GRO) is a widely used SW-based offloading technique to reduce per-packet processing overheads. By reassembling small packets into larger ones, GRO enables applications to process fewer large packets directly, thus reducing the number of packets to be processed. To benefit DPDK-based applications, like Open vSwitch, DPDK also provides own GRO implementation. In DPDK, GRO is implemented as a standalone library. Applications explicitly use the GRO library to reassemble packets.

5.32.1 Overview

In the GRO library, there are many GRO types which are defined by packet types. One GRO type is in charge of process one kind of packets. For example, TCP/IPv4 GRO processes TCP/IPv4 packets.

Each GRO type has a reassembly function, which defines own algorithm and table structure to reassemble packets. We assign input packets to the corresponding GRO functions by MBUF->packet_type.

The GRO library doesn't check if input packets have correct checksums and doesn't re-calculate checksums for merged packets. The GRO library assumes the packets are complete (i.e., MF==0 && frag_off==0), when IP fragmentation is possible (i.e., DF==0). Additionally, it complies RFC 6864 to process the IPv4 ID field.

Currently, the GRO library provides GRO supports for TCP/IPv4 packets and VxLAN packets which contain an outer IPv4 header and an inner TCP/IPv4 packet.

5.32.2 Two Sets of API

For different usage scenarios, the GRO library provides two sets of API. The one is called the lightweight mode API, which enables applications to merge a small number of packets rapidly; the other is called the heavyweight mode API, which provides fine-grained controls to applications and supports to merge a large number of packets.

Lightweight Mode API

The lightweight mode only has one function `rte_gro_reassemble_burst()`, which process N packets at a time. Using the lightweight mode API to merge packets is very simple. Calling `rte_gro_reassemble_burst()` is enough. The GROed packets are returned to applications as soon as it finishes.

In `rte_gro_reassemble_burst()`, table structures of different GRO types are allocated in the stack. This design simplifies applications' operations. However, limited by the stack size, the maximum number of packets that `rte_gro_reassemble_burst()` can process in an invocation should be less than or equal to `RTE_GRO_MAX_BURST_ITEM_NUM`.

Heavyweight Mode API

Compared with the lightweight mode, using the heavyweight mode API is relatively complex. Firstly, applications need to create a GRO context by `rte_gro_ctx_create()`. `rte_gro_ctx_create()` allocates tables structures in the heap and stores their pointers in the GRO context. Secondly, applications use `rte_gro_reassemble()` to merge packets. If input packets have invalid parameters, `rte_gro_reassemble()` returns them to applications. For example, packets of unsupported GRO types or TCP SYN packets are returned. Otherwise, the input packets are either merged with the existed packets in the tables or inserted into the tables. Finally, applications use `rte_gro_timeout_flush()` to flush packets from the tables, when they want to get the GROed packets.

Note that all update/lookup operations on the GRO context are not thread safe. So if different processes or threads want to access the same context object simultaneously, some external syncing mechanisms must be used.

5.32.3 Reassembly Algorithm

The reassembly algorithm is used for reassembling packets. In the GRO library, different GRO types can use different algorithms. In this section, we will introduce an algorithm, which is used by TCP/IPv4 GRO and VxLAN GRO.

Challenges

The reassembly algorithm determines the efficiency of GRO. There are two challenges in the algorithm design:

- a high cost algorithm/implementation would cause packet dropping in a high speed network.
- packet reordering makes it hard to merge packets. For example, Linux GRO fails to merge packets when encounters packet reordering.

The above two challenges require our algorithm is:

- lightweight enough to scale fast networking speed
- capable of handling packet reordering

In DPDK GRO, we use a key-based algorithm to address the two challenges.

Key-based Reassembly Algorithm

Fig. 5.59 illustrates the procedure of the key-based algorithm. Packets are classified into "flows" by some header fields (we call them as "key"). To process an input packet, the algorithm searches for a matched "flow" (i.e., the same value of key) for the packet first, then checks all packets in the "flow" and tries to find a "neighbor" for it. If find a "neighbor", merge the two packets together. If can't find a "neighbor", store the packet into its "flow". If can't find a matched "flow", insert a new "flow" and store the packet into the "flow".

Note: Packets in the same "flow" that can't merge are always caused by packet reordering.

The key-based algorithm has two characters:

- classifying packets into "flows" to accelerate packet aggregation is simple (address challenge 1).
- storing out-of-order packets makes it possible to merge later (address challenge 2).

Fig. 5.59: Key-based Reassembly Algorithm

5.32.4 TCP/IPv4 GRO

The table structure used by TCP/IPv4 GRO contains two arrays: flow array and item array. The flow array keeps flow information, and the item array keeps packet information.

Header fields used to define a TCP/IPv4 flow include:

- source and destination: Ethernet and IP address, TCP port
- TCP acknowledge number

TCP/IPv4 packets whose FIN, SYN, RST, URG, PSH, ECE or CWR bit is set won't be processed.

Header fields deciding if two packets are neighbors include:

- TCP sequence number
- IPv4 ID. The IPv4 ID fields of the packets, whose DF bit is 0, should be increased by 1.

5.32.5 VxLAN GRO

The table structure used by VxLAN GRO, which is in charge of processing VxLAN packets with an outer IPv4 header and inner TCP/IPv4 packet, is similar with that of TCP/IPv4 GRO. Differently, the header fields used to define a VxLAN flow include:

- outer source and destination: Ethernet and IP address, UDP port
- VxLAN header (VNI and flag)
- inner source and destination: Ethernet and IP address, TCP port

Header fields deciding if packets are neighbors include:

- outer IPv4 ID. The IPv4 ID fields of the packets, whose DF bit in the outer IPv4 header is 0, should be increased by 1.
- inner TCP sequence number
- inner IPv4 ID. The IPv4 ID fields of the packets, whose DF bit in the inner IPv4 header is 0, should be increased by 1.

Note: We comply RFC 6864 to process the IPv4 ID field. Specifically, we check IPv4 ID fields for the packets whose DF bit is 0 and ignore IPv4 ID fields for the packets whose DF bit is 1. Additionally, packets which have different value of DF bit can't be merged.

5.32.6 GRO Library Limitations

- GRO library uses MBUF->l2_len/l3_len/l4_len/outer_l2_len/ outer_l3_len/packet_type to get protocol headers for the input packet, rather than parsing the packet header. Therefore, before call GRO APIs to merge packets, user applications must set MBUF->l2_len/l3_len/l4_len/outer_l2_len/outer_l3_len/ packet_type to the same values as the protocol headers of the packet.
- GRO library doesn't support to process the packets with IPv4 Options or VLAN tagged.
- GRO library just supports to process the packet organized in a single MBUF. If the input packet consists of multiple MBUFs (i.e. chained MBUFs), GRO reassembly behaviors are unknown.

5.33 Generic Segmentation Offload Library

5.33.1 Overview

Generic Segmentation Offload (GSO) is a widely used software implementation of TCP Segmentation Offload (TSO), which reduces per-packet processing overhead. Much like TSO, GSO gains performance by enabling upper layer applications to process a smaller number of large packets (e.g. MTU size of 64KB), instead of processing higher numbers of small packets (e.g. MTU size of 1500B), thus reducing per-packet overhead.

For example, GSO allows guest kernel stacks to transmit over-sized TCP segments that far exceed the kernel interface's MTU; this eliminates the need to segment packets within the guest, and improves the data-to-overhead ratio of both the guest-host link, and PCI bus. The expectation of the guest network

stack in this scenario is that segmentation of egress frames will take place either in the NIC HW, or where that hardware capability is unavailable, either in the host application, or network stack.

Bearing that in mind, the GSO library enables DPDK applications to segment packets in software. Note however, that GSO is implemented as a standalone library, and not via a 'fallback' mechanism (i.e. for when TSO is unsupported in the underlying hardware); that is, applications must explicitly invoke the GSO library to segment packets. The size of GSO segments `(segsz)` is configurable by the application.

5.33.2 Limitations

1. The GSO library doesn't check if input packets have correct checksums.
2. In addition, the GSO library doesn't re-calculate checksums for segmented packets (that task is left to the application).
3. IP fragments are unsupported by the GSO library.
4. The egress interface's driver must support multi-segment packets.
5. Currently, the GSO library supports the following IPv4 packet types:
   - TCP
   - UDP
   - VxLAN
   - GRE

   See *Supported GSO Packet Types* for further details.

5.33.3 Packet Segmentation

The `rte_gso_segment()` function is the GSO library's primary segmentation API.

Before performing segmentation, an application must create a GSO context object `(struct rte_gso_ctx)`, which provides the library with some of the information required to understand how the packet should be segmented. Refer to *How to Segment a Packet* for additional details on same. Once the GSO context has been created, and populated, the application can then use the `rte_gso_segment()` function to segment packets.

The GSO library typically stores each segment that it creates in two parts: the first part contains a copy of the original packet's headers, while the second part contains a pointer to an offset within the original packet. This mechanism is explained in more detail in *GSO Output Segment Format*.

The GSO library supports both single- and multi-segment input mbufs.

GSO Output Segment Format

To reduce the number of expensive memcpy operations required when segmenting a packet, the GSO library typically stores each segment that it creates as a two-part mbuf (technically, this is termed a 'two-segment' mbuf; however, since the elements produced by the API are also called 'segments', for clarity the term 'part' is used here instead).

The first part of each output segment is a direct mbuf and contains a copy of the original packet's headers, which must be prepended to each output segment. These headers are copied from the original packet into each output segment.

The second part of each output segment, represents a section of data from the original packet, i.e. a data segment. Rather than copy the data directly from the original packet into the output segment (which would impact performance considerably), the second part of each output segment is an indirect mbuf, which contains no actual data, but simply points to an offset within the original packet.

The combination of the 'header' segment and the 'data' segment constitutes a single logical output GSO segment of the original packet. This is illustrated in Fig. 5.60.

Fig. 5.60: Two-part GSO output segment

In one situation, the output segment may contain additional 'data' segments. This only occurs when:

- the input packet on which GSO is to be performed is represented by a multi-segment mbuf.
- the output segment is required to contain data that spans the boundaries between segments of the input multi-segment mbuf.

The GSO library traverses each segment of the input packet, and produces numerous output segments; for optimal performance, the number of output segments is kept to a minimum. Consequently, the GSO library maximizes the amount of data contained within each output segment; i.e. each output segment `segsz` bytes of data. The only exception to this is in the case of the very final output segment; if `pkt_len` % `segsz`, then the final segment is smaller than the rest.

In order for an output segment to meet its MSS, it may need to include data from multiple input segments. Due to the nature of indirect mbufs (each indirect mbuf can point to only one direct mbuf), the solution here is to add another indirect mbuf to the output segment; this additional segment then points to the next input segment. If necessary, this chaining process is repeated, until the sum of all of the data 'contained' in the output segment reaches `segsz`. This ensures that the amount of data contained within each output segment is uniform, with the possible exception of the last segment, as previously described.

Fig. 5.61 illustrates an example of a three-part output segment. In this example, the output segment needs to include data from the end of one input segment, and the beginning of another. To achieve this, an additional indirect mbuf is chained to the second part of the output segment, and is attached to the next input segment (i.e. it points to the data in the next input segment).

Fig. 5.61: Three-part GSO output segment

5.33.4 Supported GSO Packet Types

TCP/IPv4 GSO

TCP/IPv4 GSO supports segmentation of suitably large TCP/IPv4 packets, which may also contain an optional VLAN tag.

UDP/IPv4 GSO

UDP/IPv4 GSO supports segmentation of suitably large UDP/IPv4 packets, which may also contain an optional VLAN tag. UDP GSO is the same as IP fragmentation. Specifically, UDP GSO treats the UDP header as a part of the payload and does not modify it during segmentation. Therefore, after UDP GSO, only the first output packet has the original UDP header, and others just have l2 and l3 headers.

VxLAN GSO

VxLAN packets GSO supports segmentation of suitably large VxLAN packets, which contain an outer IPv4 header, inner TCP/IPv4 headers, and optional inner and/or outer VLAN tag(s).

GRE GSO

GRE GSO supports segmentation of suitably large GRE packets, which contain an outer IPv4 header, inner TCP/IPv4 headers, and an optional VLAN tag.

5.33.5 How to Segment a Packet

To segment an outgoing packet, an application must:

1. First create a GSO context (`struct rte_gso_ctx`); this contains:
   - a pointer to the mbuf pool for allocating the direct buffers, which are used to store the GSO segments' packet headers.
   - a pointer to the mbuf pool for allocating indirect buffers, which are used to locate GSO segments' packet payloads.

     Note: An application may use the same pool for both direct and indirect buffers. However, since indirect mbufs simply store a pointer, the application may reduce its memory consumption by creating a separate memory pool, containing smaller elements, for the indirect pool.

   - the size of each output segment, including packet headers and payload, measured in bytes.
   - the bit mask of required GSO types. The GSO library uses the same macros as those that describe a physical device's TX offloading capabilities (i.e. `DEV_TX_OFFLOAD_*_TSO`) for gso_types. For example, if an application wants to segment TCP/IPv4 packets, it should set gso_types to `DEV_TX_OFFLOAD_TCP_TSO`. The only other supported values currently supported for gso_types are `DEV_TX_OFFLOAD_VXLAN_TNL_TSO`, and `DEV_TX_OFFLOAD_GRE_TNL_TSO`; a combination of these macros is also allowed.

- a flag, that indicates whether the IPv4 headers of output segments should contain fixed or incremental ID values.

2. Set the appropriate ol_flags in the mbuf.
   - The GSO library use the value of an mbuf's `ol_flags` attribute to determine how a packet should be segmented. It is the application's responsibility to ensure that these flags are set.
   - For example, in order to segment TCP/IPv4 packets, the application should add the `PKT_TX_IPV4` and `PKT_TX_TCP_SEG` flags to the mbuf's ol_flags.
   - If checksum calculation in hardware is required, the application should also add the `PKT_TX_TCP_CKSUM` and `PKT_TX_IP_CKSUM` flags.
3. Check if the packet should be processed. Packets with one of the following properties are not processed and are returned immediately:
   - Packet length is less than `segsz` (i.e. GSO is not required).
   - Packet type is not supported by GSO library (see *Supported GSO Packet Types*).
   - Application has not enabled GSO support for the packet type.
   - Packet's ol_flags have been incorrectly set.
4. Allocate space in which to store the output GSO segments. If the amount of space allocated by the application is insufficient, segmentation will fail.
5. Invoke the GSO segmentation API, `rte_gso_segment()`.
6. If required, update the L3 and L4 checksums of the newly-created segments. For tunneled packets, the outer IPv4 headers' checksums should also be updated. Alternatively, the application may offload checksum calculation to HW.

5.34 The librte_pdump Library

The `librte_pdump` library provides a framework for packet capturing in DPDK. The library does the complete copy of the Rx and Tx mbufs to a new mempool and hence it slows down the performance of the applications, so it is recommended to use this library for debugging purposes.

The library provides the following APIs to initialize the packet capture framework, to enable or disable the packet capture, and to uninitialize it:

- `rte_pdump_init()`: This API initializes the packet capture framework.
- `rte_pdump_enable()`: This API enables the packet capture on a given port and queue. Note: The filter option in the API is a place holder for future enhancements.
- `rte_pdump_enable_by_deviceid()`: This API enables the packet capture on a given device id (`vdev name or pci address`) and queue. Note: The filter option in the API is a place holder for future enhancements.
- `rte_pdump_disable()`: This API disables the packet capture on a given port and queue.
- `rte_pdump_disable_by_deviceid()`: This API disables the packet capture on a given device id (`vdev name or pci address`) and queue.
- `rte_pdump_uninit()`: This API uninitializes the packet capture framework.

5.34.1 Operation

The librte_pdump library works on a client/server model. The server is responsible for enabling or disabling the packet capture and the clients are responsible for requesting the enabling or disabling of the packet capture.

The packet capture framework, as part of its initialization, creates the pthread and the server socket in the pthread. The application that calls the framework initialization will have the server socket created, either under the path that the application has passed or under the default path i.e. either /var/run/.dpdk for root user or ~/.dpdk for non root user.

Applications that request enabling or disabling of the packet capture will have the client socket created either under the path that the application has passed or under the default path i.e. either /var/run/.dpdk for root user or ~/.dpdk for not root user to send the requests to the server. The server socket will listen for client requests for enabling or disabling the packet capture.

5.34.2 Implementation Details

The library API rte_pdump_init(), initializes the packet capture framework by creating the pdump server by calling rte_mp_action_register() function. The server will listen to the client requests to enable or disable the packet capture.

The library APIs rte_pdump_enable() and rte_pdump_enable_by_deviceid() enables the packet capture. On each call to these APIs, the library creates a separate client socket, creates the "pdump enable" request and sends the request to the server. The server that is listening on the socket will take the request and enable the packet capture by registering the Ethernet RX and TX callbacks for the given port or device_id and queue combinations. Then the server will mirror the packets to the new mempool and enqueue them to the rte_ring that clients have passed to these APIs. The server also sends the response back to the client about the status of the request that was processed. After the response is received from the server, the client socket is closed.

The library APIs rte_pdump_disable() and rte_pdump_disable_by_deviceid() disables the packet capture. On each call to these APIs, the library creates a separate client socket, creates the "pdump disable" request and sends the request to the server. The server that is listening on the socket will take the request and disable the packet capture by removing the Ethernet RX and TX callbacks for the given port or device_id and queue combinations. The server also sends the response back to the client about the status of the request that was processed. After the response is received from the server, the client socket is closed.

The library API rte_pdump_uninit(), uninitializes the packet capture framework by calling rte_mp_action_unregister() function.

5.34.3 Use Case: Packet Capturing

The DPDK app/pdump tool is developed based on this library to capture packets in DPDK. Users can use this as an example to develop their own packet capturing tools.

5.35 Multi-process Support

In the DPDK, multi-process support is designed to allow a group of DPDK processes to work together in a simple transparent manner to perform packet processing, or other workloads. To support this functionality, a number of additions have been made to the core DPDK Environment Abstraction Layer (EAL).

The EAL has been modified to allow different types of DPDK processes to be spawned, each with different permissions on the hugepage memory used by the applications. For now, there are two types of process specified:

- primary processes, which can initialize and which have full permissions on shared memory
- secondary processes, which cannot initialize shared memory, but can attach to pre- initialized shared memory and create objects in it.

Standalone DPDK processes are primary processes, while secondary processes can only run alongside a primary process or after a primary process has already configured the hugepage shared memory for them.

Note: Secondary processes should run alongside primary process with same DPDK version.

Secondary processes which requires access to physical devices in Primary process, must be passed with the same whitelist and blacklist options.

To support these two process types, and other multi-process setups described later, two additional command-line parameters are available to the EAL:

- `--proc-type:` for specifying a given process instance as the primary or secondary DPDK instance
- `--file-prefix:` to allow processes that do not want to co-operate to have different memory regions

A number of example applications are provided that demonstrate how multiple DPDK processes can be used together. These are more fully documented in the "Multi- process Sample Application" chapter in the *DPDK Sample Application's User Guide*.

5.35.1 Memory Sharing

The key element in getting a multi-process application working using the DPDK is to ensure that memory resources are properly shared among the processes making up the multi-process application. Once there are blocks of shared memory available that can be accessed by multiple processes, then issues such as inter-process communication (IPC) becomes much simpler.

On application start-up in a primary or standalone process, the DPDK records to memory-mapped files the details of the memory configuration it is using - hugepages in use, the virtual addresses they are mapped at, the number of memory channels present, etc. When a secondary process is started, these files are read and the EAL recreates the same memory configuration in the secondary process so that all memory zones are shared between processes and all pointers to that memory are valid, and point to the same objects, in both processes.

Note: Refer to *Multi-process Limitations* for details of how Linux kernel Address-Space Layout Randomization (ASLR) can affect memory sharing.

If the primary process was run with `--legacy-mem` or `--single-file-segments` switch, secondary processes must be run with the same switch specified. Otherwise, memory corruption may occur.

Fig. 5.62: Memory Sharing in the DPDK Multi-process Sample Application

The EAL also supports an auto-detection mode (set by EAL `--proc-type=auto` flag), whereby an DPDK process is started as a secondary instance if a primary instance is already running.

5.35.2 Deployment Models

Symmetric/Peer Processes

DPDK multi-process support can be used to create a set of peer processes where each process performs the same workload. This model is equivalent to having multiple threads each running the same main-loop function, as is done in most of the supplied DPDK sample applications. In this model, the first of the processes spawned should be spawned using the `--proc-type=primary` EAL flag, while all subsequent instances should be spawned using the `--proc-type=secondary` flag.

The simple_mp and symmetric_mp sample applications demonstrate this usage model. They are described in the "Multi-process Sample Application" chapter in the *DPDK Sample Application's User Guide*.

Asymmetric/Non-Peer Processes

An alternative deployment model that can be used for multi-process applications is to have a single primary process instance that acts as a load-balancer or server distributing received packets among worker or client threads, which are run as secondary processes. In this case, extensive use of rte_ring objects is made, which are located in shared hugepage memory.

The client_server_mp sample application shows this usage model. It is described in the "Multi-process Sample Application" chapter in the *DPDK Sample Application's User Guide*.

Running Multiple Independent DPDK Applications

In addition to the above scenarios involving multiple DPDK processes working together, it is possible to run multiple DPDK processes side-by-side, where those processes are all working independently. Support for this usage scenario is provided using the `--file-prefix` parameter to the EAL.

By default, the EAL creates hugepage files on each hugetlbfs filesystem using the rtemap_X filename, where X is in the range 0 to the maximum number of hugepages -1. Similarly, it creates shared configuration files, memory mapped in each process, using the /var/run/.rte_config filename, when run as root (or $HOME/.rte_config when run as a non-root user; if filesystem and device permissions are set up to allow this). The rte part of the filenames of each of the above is configurable using the file-prefix parameter.

In addition to specifying the file-prefix parameter, any DPDK applications that are to be run side-by-side must explicitly limit their memory use. This is less of a problem on Linux, as by default, applications will not allocate more memory than they need. However if `--legacy-mem` is used, DPDK will attempt to preallocate all memory it can get to, and memory use must be explicitly limited. This is done by passing the `-m` flag to each process to specify how much hugepage memory, in megabytes, each process can use

(or passing `--socket-mem` to specify how much hugepage memory on each socket each process can use).

Note: Independent DPDK instances running side-by-side on a single machine cannot share any network ports. Any network ports being used by one process should be blacklisted in every other process.

Running Multiple Independent Groups of DPDK Applications

In the same way that it is possible to run independent DPDK applications side- by-side on a single system, this can be trivially extended to multi-process groups of DPDK applications running side-by-side. In this case, the secondary processes must use the same `--file-prefix` parameter as the primary process whose shared memory they are connecting to.

Note: All restrictions and issues with multiple independent DPDK processes running side-by-side apply in this usage scenario also.

5.35.3 Multi-process Limitations

There are a number of limitations to what can be done when running DPDK multi-process applications. Some of these are documented below:

- The multi-process feature requires that the exact same hugepage memory mappings be present in all applications. This makes secondary process startup process generally unreliable. Disabling Linux security feature - Address-Space Layout Randomization (ASLR) may help getting more consistent mappings, but not necessarily more reliable - if the mappings are wrong, they will be consistently wrong!

Warning: Disabling Address-Space Layout Randomization (ASLR) may have security implications, so it is recommended that it be disabled only when absolutely necessary, and only when the implications of this change have been understood.

- All DPDK processes running as a single application and using shared memory must have distinct coremask/corelist arguments. It is not possible to have a primary and secondary instance, or two secondary instances, using any of the same logical cores. Attempting to do so can cause corruption of memory pool caches, among other issues.
- The delivery of interrupts, such as Ethernet* device link status interrupts, do not work in secondary processes. All interrupts are triggered inside the primary process only. Any application needing interrupt notification in multiple processes should provide its own mechanism to transfer the interrupt information from the primary process to any secondary process that needs the information.
- The use of function pointers between multiple processes running based of different compiled binaries is not supported, since the location of a given function in one process may be different to its location in a second. This prevents the librte_hash library from behaving properly as in a multi-process instance, since it uses a pointer to the hash function internally.

To work around this issue, it is recommended that multi-process applications perform the hash calculations by directly calling the hashing function from the code and then using the

rte_hash_add_with_hash()/rte_hash_lookup_with_hash() functions instead of the functions which do the hashing internally, such as rte_hash_add()/rte_hash_lookup().

- Depending upon the hardware in use, and the number of DPDK processes used, it may not be possible to have HPET timers available in each DPDK instance. The minimum number of HPET comparators available to Linux* userspace can be just a single comparator, which means that only the first, primary DPDK process instance can open and mmap /dev/hpet. If the number of required DPDK processes exceeds that of the number of available HPET comparators, the TSC (which is the default timer in this release) must be used as a time source across all processes instead of the HPET.

5.35.4 Communication between multiple processes

While there are multiple ways one can approach inter-process communication in DPDK, there is also a native DPDK IPC API available. It is not intended to be performance-critical, but rather is intended to be a convenient, general purpose API to exchange short messages between primary and secondary processes.

DPDK IPC API supports the following communication modes:

- Unicast message from secondary to primary
- Broadcast message from primary to all secondaries

In other words, any IPC message sent in a primary process will be delivered to all secondaries, while any IPC message sent in a secondary process will only be delivered to primary process. Unicast from primary to secondary or from secondary to secondary is not supported.

There are three types of communications that are available within DPDK IPC API:

- Message
- Synchronous request
- Asynchronous request

A "message" type does not expect a response and is meant to be a best-effort notification mechanism, while the two types of "requests" are meant to be a two way communication mechanism, with the requester expecting a response from the other side.

Both messages and requests will trigger a named callback on the receiver side. These callbacks will be called from within a dedicated IPC or interrupt thread that are not part of EAL lcore threads.

Registering for incoming messages

Before any messages can be received, a callback will need to be registered. This is accomplished by calling `rte_mp_action_register()` function. This function accepts a unique callback name, and a function pointer to a callback that will be called when a message or a request matching this callback name arrives.

If the application is no longer willing to receive messages intended for a specific callback function, `rte_mp_action_unregister()` function can be called to ensure that callback will not be triggered again.

Sending messages

To send a message, a `rte_mp_msg` descriptor must be populated first. The list of fields to be populated are as follows:

- `name` - message name. This name must match receivers' callback name.
- `param` - message data (up to 256 bytes).
- `len_param` - length of message data.
- `fds` - file descriptors to pass long with the data (up to 8 fd's).
- `num_fds` - number of file descriptors to send.

Once the structure is populated, calling `rte_mp_sendmsg()` will send the descriptor either to all secondary processes (if sent from primary process), or to primary process (if sent from secondary process). The function will return a value indicating whether sending the message succeeded or not.

Sending requests

Sending requests involves waiting for the other side to reply, so they can block for a relatively long time.

To send a request, a message descriptor `rte_mp_msg` must be populated. Additionally, a `timespec` value must be specified as a timeout, after which IPC will stop waiting and return.

For synchronous requests, the `rte_mp_reply` descriptor must also be created. This is where the responses will be stored. The list of fields that will be populated by IPC are as follows:

- `nb_sent` - number indicating how many requests were sent (i.e. how many peer processes were active at the time of the request).
- `nb_received` - number indicating how many responses were received (i.e. of those peer processes that were active at the time of request, how many have replied)
- `msgs` - pointer to where all of the responses are stored. The order in which responses appear is undefined. When doing synchronous requests, this memory must be freed by the requestor after request completes!

For asynchronous requests, a function pointer to the callback function must be provided instead. This callback will be called when the request either has timed out, or will have received a response to all the messages that were sent.

> **Warning:** When an asynchronous request times out, the callback will be called not by a dedicated IPC thread, but rather from EAL interrupt thread. Because of this, it may not be possible for DPDK to trigger another interrupt-based event (such as an alarm) while handling asynchronous IPC callback.

When the callback is called, the original request descriptor will be provided (so that it would be possible to determine for which sent message this is a callback to), along with a response descriptor like the one described above. When doing asynchronous requests, there is no need to free the resulting `rte_mp_reply` descriptor.

Receiving and responding to messages

To receive a message, a name callback must be registered using the `rte_mp_action_register()` function. The name of the callback must match the `name` field in sender's `rte_mp_msg` message descriptor in order for this message to be delivered and for the callback to be trigger.

The callback's definition is `rte_mp_t`, and consists of the incoming message pointer `msg`, and an opaque pointer `peer`. Contents of `msg` will be identical to ones sent by the sender.

If a response is required, a new `rte_mp_msg` message descriptor must be constructed and sent via `rte_mp_reply()` function, along with `peer` pointer. The resulting response will then be delivered to the correct requestor.

Warning: Simply returning a value when processing a request callback will not send a response to the request - it must always be explicitly sent even in case of errors. Implementation of error signalling rests with the application, there is no built-in way to indicate success or error for a request. Failing to do so will cause the requestor to time out while waiting on a response.

Misc considerations

Due to the underlying IPC implementation being single-threaded, recursive requests (i.e. sending a request while responding to another request) is not supported. However, since sending messages (not requests) does not involve an IPC thread, sending messages while processing another message or request is supported.

Since the memory sybsystem uses IPC internally, memory allocations and IPC must not be mixed: it is not safe to use IPC inside a memory-related callback, nor is it safe to allocate/free memory inside IPC callbacks. Attempting to do so may lead to a deadlock.

Asynchronous request callbacks may be triggered either from IPC thread or from interrupt thread, depending on whether the request has timed out. It is therefore suggested to avoid waiting for interrupt-based events (such as alarms) inside asynchronous IPC request callbacks. This limitation does not apply to messages or synchronous requests.

If callbacks spend a long time processing the incoming requests, the requestor might time out, so setting the right timeout value on the requestor side is imperative.

If some of the messages timed out, `nb_sent` and `nb_received` fields in the `rte_mp_reply` descriptor will not have matching values. This is not treated as error by the IPC API, and it is expected that the user will be responsible for deciding how to handle such cases.

If a callback has been registered, IPC will assume that it is safe to call it. This is important when registering callbacks during DPDK initialization. During initialization, IPC will consider the receiving side as non-existing if the callback has not been registered yet. However, once the callback has been registered, it is expected that IPC should be safe to trigger it, even if the rest of the DPDK initialization hasn't finished yet.

5.36 Kernel NIC Interface

The DPDK Kernel NIC Interface (KNI) allows userspace applications access to the Linux* control plane.

The benefits of using the DPDK KNI are:

- Faster than existing Linux TUN/TAP interfaces (by eliminating system calls and copy_to_user()/copy_from_user() operations.
- Allows management of DPDK ports using standard Linux net tools such as ethtool, ifconfig and tcpdump.
- Allows an interface with the kernel network stack.

The components of an application using the DPDK Kernel NIC Interface are shown in Fig. 5.63.

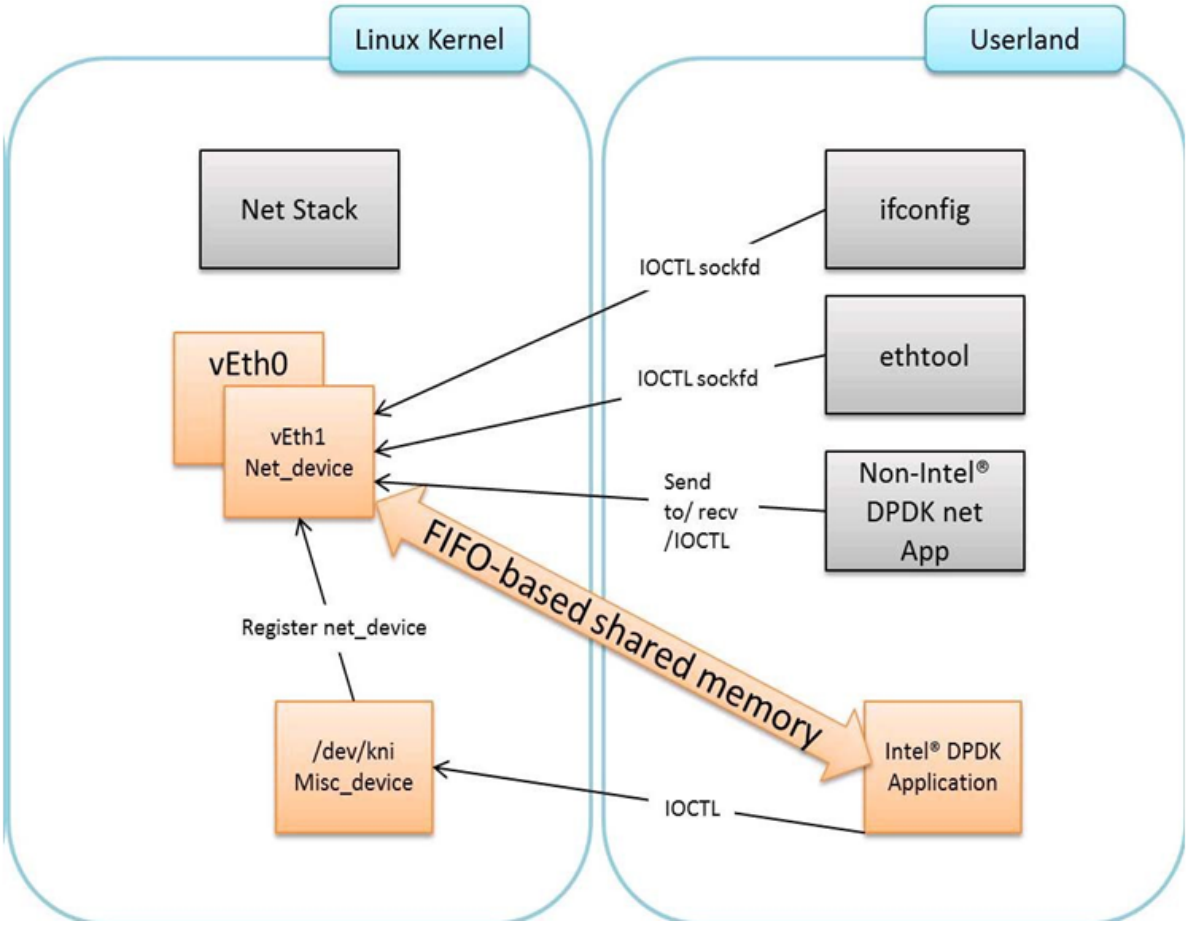


Fig. 5.63: Components of a DPDK KNI Application

5.36.1 The DPDK KNI Kernel Module

The KNI kernel loadable module `rte_kni` provides the kernel interface for DPDK applications.

When the `rte_kni` module is loaded, it will create a device `/dev/kni` that is used by the DPDK KNI API functions to control and communicate with the kernel module.

The `rte_kni` kernel module contains several optional parameters which can be specified when the module is loaded to control its behavior:

```
# modinfo rte_kni.ko
<snip>
parm:           lo_mode: KNI loopback mode (default=lo_mode_none):
                lo_mode_none        Kernel loopback disabled
                lo_mode_fifo        Enable kernel loopback with fifo
                lo_mode_fifo_skb    Enable kernel loopback with fifo and skb buffer
                 (charp)
parm:           kthread_mode: Kernel thread mode (default=single):
                single    Single kernel thread mode enabled.
                multiple  Multiple kernel thread mode enabled.
                 (charp)
parm:           carrier: Default carrier state for KNI interface (default=off):
                off   Interfaces will be created with carrier state set to off.
                on    Interfaces will be created with carrier state set to on.
                 (charp)
```

Loading the `rte_kni` kernel module without any optional parameters is the typical way a DPDK application gets packets into and out of the kernel network stack. Without any parameters, only one kernel thread is created for all KNI devices for packet receiving in kernel side, loopback mode is disabled, and the default carrier state of KNI interfaces is set to *off*.

```
# insmod kmod/rte_kni.ko
```

Loopback Mode

For testing, the `rte_kni` kernel module can be loaded in loopback mode by specifying the `lo_mode` parameter:

```
# insmod kmod/rte_kni.ko lo_mode=lo_mode_fifo
```

The `lo_mode_fifo` loopback option will loop back ring enqueue/dequeue operations in kernel space.

```
# insmod kmod/rte_kni.ko lo_mode=lo_mode_fifo_skb
```

The `lo_mode_fifo_skb` loopback option will loop back ring enqueue/dequeue operations and sk buffer copies in kernel space.

If the `lo_mode` parameter is not specified, loopback mode is disabled.

Kernel Thread Mode

To provide flexibility of performance, the `rte_kni` KNI kernel module can be loaded with the `kthread_mode` parameter. The `rte_kni` kernel module supports two options: "single kernel thread" mode and "multiple kernel thread" mode.

Single kernel thread mode is enabled as follows:

```
# insmod kmod/rte_kni.ko kthread_mode=single
```

This mode will create only one kernel thread for all KNI interfaces to receive data on the kernel side. By default, this kernel thread is not bound to any particular core, but the user can set the core affinity for this kernel thread by setting the `core_id` and `force_bind` parameters in `struct rte_kni_conf` when the first KNI interface is created:

For optimum performance, the kernel thread should be bound to a core in on the same socket as the DPDK lcores used in the application.

The KNI kernel module can also be configured to start a separate kernel thread for each KNI interface created by the DPDK application. Multiple kernel thread mode is enabled as follows:

```
# insmod kmod/rte_kni.ko kthread_mode=multiple
```

This mode will create a separate kernel thread for each KNI interface to receive data on the kernel side. The core affinity of each `kni_thread` kernel thread can be specified by setting the `core_id` and `force_bind` parameters in `struct rte_kni_conf` when each KNI interface is created.

Multiple kernel thread mode can provide scalable higher performance if sufficient unused cores are available on the host system.

If the `kthread_mode` parameter is not specified, the "single kernel thread" mode is used.

Default Carrier State

The default carrier state of KNI interfaces created by the `rte_kni` kernel module is controlled via the `carrier` option when the module is loaded.

If `carrier=off` is specified, the kernel module will leave the carrier state of the interface *down* when the interface is management enabled. The DPDK application can set the carrier state of the KNI interface using the `rte_kni_update_link()` function. This is useful for DPDK applications which require that the carrier state of the KNI interface reflect the actual link state of the corresponding physical NIC port.

If `carrier=on` is specified, the kernel module will automatically set the carrier state of the interface to *up* when the interface is management enabled. This is useful for DPDK applications which use the KNI interface as a purely virtual interface that does not correspond to any physical hardware and do not wish to explicitly set the carrier state of the interface with `rte_kni_update_link()`. It is also useful for testing in loopback mode where the NIC port may not be physically connected to anything.

To set the default carrier state to *on*:

```
# insmod kmod/rte_kni.ko carrier=on
```

To set the default carrier state to *off*:

```
# insmod kmod/rte_kni.ko carrier=off
```

If the `carrier` parameter is not specified, the default carrier state of KNI interfaces will be set to *off*.

5.36.2 KNI Creation and Deletion

Before any KNI interfaces can be created, the rte_kni kernel module must be loaded into the kernel and configured withe rte_kni_init() function.

The KNI interfaces are created by a DPDK application dynamically via the rte_kni_alloc() function.

The struct rte_kni_conf structure contains fields which allow the user to specify the interface name, set the MTU size, set an explicit or random MAC address and control the affinity of the kernel Rx thread(s) (both single and multi-threaded modes). By default the KNI sample example gets the MTU from the matching device, and in case of KNI PMD it is derived from mbuf buffer length.

The struct rte_kni_ops structure contains pointers to functions to handle requests from the rte_kni kernel module. These functions allow DPDK applications to perform actions when the KNI interfaces are manipulated by control commands or functions external to the application.

For example, the DPDK application may wish to enabled/disable a physical NIC port when a user enabled/disables a KNI interface with ip link set [up|down] dev <ifaceX>. The DPDK application can register a callback for config_network_if which will be called when the interface management state changes.

There are currently four callbacks for which the user can register application functions:

config_network_if:

> Called when the management state of the KNI interface changes. For example, when the user runs ip link set [up|down] dev <ifaceX>.

change_mtu:

> Called when the user changes the MTU size of the KNI interface. For example, when the user runs ip link set mtu <size> dev <ifaceX>.

config_mac_address:

> Called when the user changes the MAC address of the KNI interface. For example, when the user runs ip link set address <MAC> dev <ifaceX>. If the user sets this callback function to NULL, but sets the port_id field to a value other than -1, a default callback handler in the rte_kni library kni_config_mac_address() will be called which calls rte_eth_dev_default_mac_addr_set() on the specified port_id.

config_promiscusity:

> Called when the user changes the promiscuity state of the KNI interface. For example, when the user runs ip link set promisc [on|off] dev <ifaceX>. If the user sets this callback function to NULL, but sets the port_id field to a value other than -1, a default callback handler in the rte_kni library kni_config_promiscusity() will be called which calls rte_eth_promiscuous_<enable|disable>() on the specified port_id.

config_allmulticast:

> Called when the user changes the allmulticast state of the KNI interface. For example, when the user runs ifconfig <ifaceX> [-]allmulti. If the user sets this callback function to NULL, but sets the port_id field to a value other than -1, a default callback handler in the rte_kni library kni_config_allmulticast() will be called which calls rte_eth_allmulticast_<enable|disable>() on the specified port_id.

In order to run these callbacks, the application must periodically call the rte_kni_handle_request() function. Any user callback function registered will be called directly from

`rte_kni_handle_request()` so care must be taken to prevent deadlock and to not block any DPDK fastpath tasks. Typically DPDK applications which use these callbacks will need to create a separate thread or secondary process to periodically call `rte_kni_handle_request()`.

The KNI interfaces can be deleted by a DPDK application with `rte_kni_release()`. All KNI interfaces not explicitly deleted will be deleted when the `/dev/kni` device is closed, either explicitly with `rte_kni_close()` or when the DPDK application is closed.

5.36.3 DPDK mbuf Flow

To minimize the amount of DPDK code running in kernel space, the mbuf mempool is managed in userspace only. The kernel module will be aware of mbufs, but all mbuf allocation and free operations will be handled by the DPDK application only.

Fig. 5.64 shows a typical scenario with packets sent in both directions.

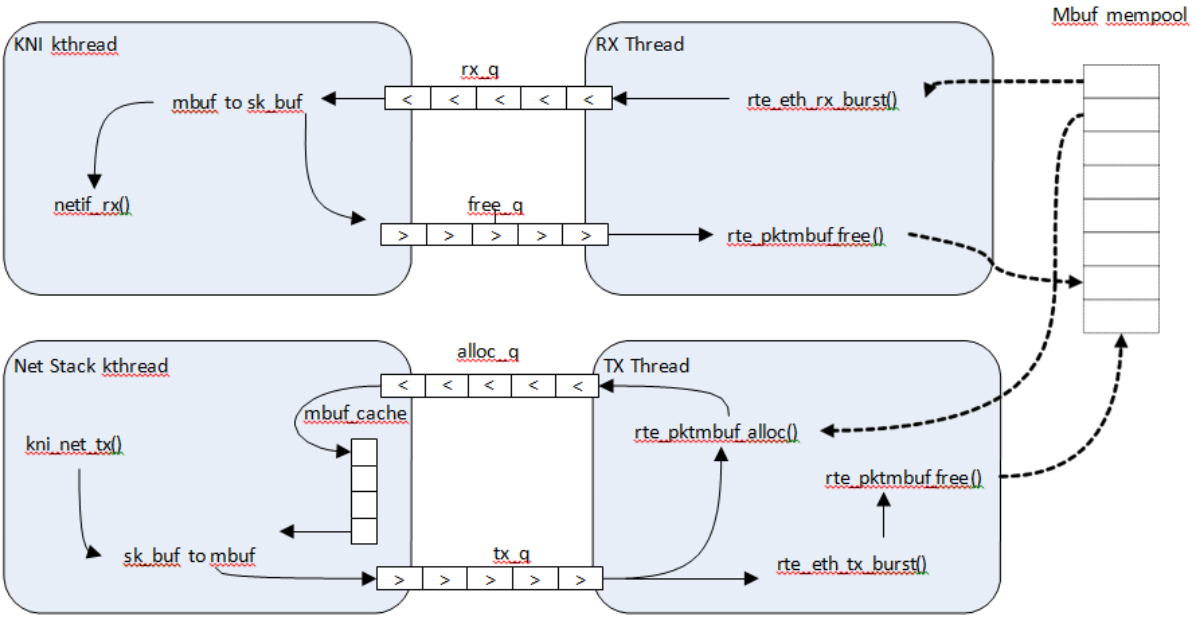


Fig. 5.64: Packet Flow via mbufs in the DPDK KNI

5.36.4 Use Case: Ingress

On the DPDK RX side, the mbuf is allocated by the PMD in the RX thread context. This thread will enqueue the mbuf in the rx_q FIFO, and the next pointers in mbuf-chain will convert to physical address. The KNI thread will poll all KNI active devices for the rx_q. If an mbuf is dequeued, it will be converted to a sk_buff and sent to the net stack via netif_rx(). The dequeued mbuf must be freed, so the same pointer is sent back in the free_q FIFO, and next pointers must convert back to virtual address if exists before put in the free_q FIFO.

The RX thread, in the same main loop, polls this FIFO and frees the mbuf after dequeuing it. The address conversion of the next pointer is to prevent the chained mbuf in different hugepage segments from causing kernel crash.

5.36.5 Use Case: Egress

For packet egress the DPDK application must first enqueue several mbufs to create an mbuf cache on the kernel side.

The packet is received from the Linux net stack, by calling the kni_net_tx() callback. The mbuf is dequeued (without waiting due the cache) and filled with data from sk_buff. The sk_buff is then freed and the mbuf sent in the tx_q FIFO.

The DPDK TX thread dequeues the mbuf and sends it to the PMD via `rte_eth_tx_burst()`. It then puts the mbuf back in the cache.

5.36.6 IOVA = VA: Support

KNI operates in IOVA_VA scheme when

- LINUX_VERSION_CODE >= KERNEL_VERSION(4, 10, 0) and
- EAL option *iova-mode=va* is passed or bus IOVA scheme in the DPDK is selected as RTE_IOVA_VA.

Due to IOVA to KVA address translations, based on the KNI use case there can be a performance impact. For mitigation, forcing IOVA to PA via EAL "–iova-mode=pa" option can be used, IOVA_DC bus iommu scheme can also result in IOVA as PA.

5.36.7 Ethtool

Ethtool is a Linux-specific tool with corresponding support in the kernel. The current version of kni provides minimal ethtool functionality including querying version and link state. It does not support link control, statistics, or dumping device registers.

5.37 Thread Safety of DPDK Functions

The DPDK is comprised of several libraries. Some of the functions in these libraries can be safely called from multiple threads simultaneously, while others cannot. This section allows the developer to take these issues into account when building their own application.

The run-time environment of the DPDK is typically a single thread per logical core. In some cases, it is not only multi-threaded, but multi-process. Typically, it is best to avoid sharing data structures between threads and/or processes where possible. Where this is not possible, then the execution blocks must access the data in a thread- safe manner. Mechanisms such as atomics or locking can be used that will allow execution blocks to operate serially. However, this can have an effect on the performance of the application.

5.37.1 Fast-Path APIs

Applications operating in the data plane are performance sensitive but certain functions within those libraries may not be safe to call from multiple threads simultaneously. The hash, LPM and mempool libraries and RX/TX in the PMD are examples of this.

The hash and LPM libraries are, by design, thread unsafe in order to maintain performance. However, if required the developer can add layers on top of these libraries to provide thread safety. Locking is not needed in all situations, and in both the hash and LPM libraries, lookups of values can be performed in parallel in multiple threads. Adding, removing or modifying values, however, cannot be done in multiple threads without using locking when a single hash or LPM table is accessed. Another alternative to locking would be to create multiple instances of these tables allowing each thread its own copy.

The RX and TX of the PMD are the most critical aspects of a DPDK application and it is recommended that no locking be used as it will impact performance. Note, however, that these functions can safely be used from multiple threads when each thread is performing I/O on a different NIC queue. If multiple threads are to use the same hardware queue on the same NIC port, then locking, or some other form of mutual exclusion, is necessary.

The ring library is based on a lockless ring-buffer algorithm that maintains its original design for thread safety. Moreover, it provides high performance for either multi- or single-consumer/producer enqueue/dequeue operations. The mempool library is based on the DPDK lockless ring library and therefore is also multi-thread safe.

5.37.2 Performance Insensitive API

Outside of the performance sensitive areas described in Section 25.1, the DPDK provides a thread-safe API for most other libraries. For example, malloc and memzone functions are safe for use in multi-threaded and multi-process environments.

The setup and configuration of the PMD is not performance sensitive, but is not thread safe either. It is possible that the multiple read/writes during PMD setup and configuration could be corrupted in a multi-thread environment. Since this is not performance sensitive, the developer can choose to add their own layer to provide thread-safe setup and configuration. It is expected that, in most applications, the initial configuration of the network ports would be done by a single thread at startup.

5.37.3 Library Initialization

It is recommended that DPDK libraries are initialized in the main thread at application startup rather than subsequently in the forwarding threads. However, the DPDK performs checks to ensure that libraries are only initialized once. If initialization is attempted more than once, an error is returned.

In the multi-process case, the configuration information of shared memory will only be initialized by the master process. Thereafter, both master and secondary processes can allocate/release any objects of memory that finally rely on rte_malloc or memzones.

5.37.4 Interrupt Thread

The DPDK works almost entirely in Linux user space in polling mode. For certain infrequent operations, such as receiving a PMD link status change notification, callbacks may be called in an additional thread outside the main DPDK processing threads. These function callbacks should avoid manipulating DPDK objects that are also managed by the normal DPDK threads, and if they need to do so, it is up to the application to provide the appropriate locking or mutual exclusion restrictions around those objects.

5.38 Event Device Library

The DPDK Event device library is an abstraction that provides the application with features to schedule events. This is achieved using the PMD architecture similar to the ethdev or cryptodev APIs, which may already be familiar to the reader.

The eventdev framework introduces the event driven programming model. In a polling model, lcores poll ethdev ports and associated Rx queues directly to look for a packet. By contrast in an event driven model, lcores call the scheduler that selects packets for them based on programmer-specified criteria. The Eventdev library adds support for an event driven programming model, which offers applications automatic multicore scaling, dynamic load balancing, pipelining, packet ingress order maintenance and synchronization services to simplify application packet processing.

By introducing an event driven programming model, DPDK can support both polling and event driven programming models for packet processing, and applications are free to choose whatever model (or combination of the two) best suits their needs.

Step-by-step instructions of the eventdev design is available in the *API Walk-through* section later in this document.

5.38.1 Event struct

The eventdev API represents each event with a generic struct, which contains a payload and metadata required for scheduling by an eventdev. The `rte_event` struct is a 16 byte C structure, defined in `libs/librte_eventdev/rte_eventdev.h`.

Event Metadata

The rte_event structure contains the following metadata fields, which the application fills in to have the event scheduled as required:

- `flow_id` - The targeted flow identifier for the enq/deq operation.
- `event_type` - The source of this event, e.g. RTE_EVENT_TYPE_ETHDEV or CPU.
- `sub_event_type` - Distinguishes events inside the application, that have the same event_type (see above)
- `op` - This field takes one of the RTE_EVENT_OP_* values, and tells the eventdev about the status of the event - valid values are NEW, FORWARD or RELEASE.
- `sched_type` - Represents the type of scheduling that should be performed on this event, valid values are the RTE_SCHED_TYPE_ORDERED, ATOMIC and PARALLEL.
- `queue_id` - The identifier for the event queue that the event is sent to.

- `priority` - The priority of this event, see RTE_EVENT_DEV_PRIORITY.

Event Payload

The rte_event struct contains a union for payload, allowing flexibility in what the actual event being scheduled is. The payload is a union of the following:

- `uint64_t u64`
- `void *event_ptr`
- `struct rte_mbuf *mbuf`

These three items in a union occupy the same 64 bits at the end of the rte_event structure. The application can utilize the 64 bits directly by accessing the u64 variable, while the event_ptr and mbuf are provided as convenience variables. For example the mbuf pointer in the union can used to schedule a DPDK packet.

Queues

An event queue is a queue containing events that are scheduled by the event device. An event queue contains events of different flows associated with scheduling types, such as atomic, ordered, or parallel.

Queue All Types Capable

If RTE_EVENT_DEV_CAP_QUEUE_ALL_TYPES capability bit is set in the event device, then events of any type may be sent to any queue. Otherwise, the queues only support events of the type that it was created with.

Queue All Types Incapable

In this case, each stage has a specified scheduling type. The application configures each queue for a specific type of scheduling, and just enqueues all events to the eventdev. An example of a PMD of this type is the eventdev software PMD.

The Eventdev API supports the following scheduling types per queue:

- Atomic
- Ordered
- Parallel

Atomic, Ordered and Parallel are load-balanced scheduling types: the output of the queue can be spread out over multiple CPU cores.

Atomic scheduling on a queue ensures that a single flow is not present on two different CPU cores at the same time. Ordered allows sending all flows to any core, but the scheduler must ensure that on egress the packets are returned to ingress order on downstream queue enqueue. Parallel allows sending all flows to all CPU cores, without any re-ordering guarantees.

Single Link Flag

There is a SINGLE_LINK flag which allows an application to indicate that only one port will be connected to a queue. Queues configured with the single-link flag follow a FIFO like structure, maintaining ordering but it is only capable of being linked to a single port (see below for port and queue linking details).

Ports

Ports are the points of contact between worker cores and the eventdev. The general use-case will see one CPU core using one port to enqueue and dequeue events from an eventdev. Ports are linked to queues in order to retrieve events from those queues (more details in *Linking Queues and Ports* below).

5.38.2 API Walk-through

This section will introduce the reader to the eventdev API, showing how to create and configure an eventdev and use it for a two-stage atomic pipeline with one core each for RX and TX. RX and TX cores are shown here for illustration, refer to Eventdev Adapter documentation for further details. The diagram below shows the final state of the application after this walk-through:

Fig. 5.65: Sample eventdev usage, with RX, two atomic stages and a single-link to TX.

A high level overview of the setup steps are:

- rte_event_dev_configure()
- rte_event_queue_setup()
- rte_event_port_setup()
- rte_event_port_link()
- rte_event_dev_start()

Init and Config

The eventdev library uses vdev options to add devices to the DPDK application. The `--vdev` EAL option allows adding eventdev instances to your DPDK application, using the name of the eventdev PMD as an argument.

For example, to create an instance of the software eventdev scheduler, the following vdev arguments should be provided to the application EAL command line:

```
./dpdk_application --vdev="event_sw0"
```

In the following code, we configure eventdev instance with 3 queues and 6 ports as follows. The 3 queues consist of 2 Atomic and 1 Single-Link, while the 6 ports consist of 4 workers, 1 RX and 1 TX.

```
const struct rte_event_dev_config config = {
        .nb_event_queues = 3,
        .nb_event_ports = 6,
        .nb_events_limit  = 4096,
```

(continues on next page)

(continued from previous page)

```
        .nb_event_queue_flows = 1024,
        .nb_event_port_dequeue_depth = 128,
        .nb_event_port_enqueue_depth = 128,
};
int err = rte_event_dev_configure(dev_id, &config);
```

The remainder of this walk-through assumes that dev_id is 0.

Setting up Queues

Once the eventdev itself is configured, the next step is to configure queues. This is done by setting the appropriate values in a queue_conf structure, and calling the setup function. Repeat this step for each queue, starting from 0 and ending at `nb_event_queues - 1` from the event_dev config above.

```
struct rte_event_queue_conf atomic_conf = {
        .schedule_type = RTE_SCHED_TYPE_ATOMIC,
        .priority = RTE_EVENT_DEV_PRIORITY_NORMAL,
        .nb_atomic_flows = 1024,
        .nb_atomic_order_sequences = 1024,
};
struct rte_event_queue_conf single_link_conf = {
        .event_queue_cfg = RTE_EVENT_QUEUE_CFG_SINGLE_LINK,
};
int dev_id = 0;
int atomic_q_1 = 0;
int atomic_q_2 = 1;
int single_link_q = 2;
int err = rte_event_queue_setup(dev_id, atomic_q_1, &atomic_conf);
int err = rte_event_queue_setup(dev_id, atomic_q_2, &atomic_conf);
int err = rte_event_queue_setup(dev_id, single_link_q, &single_link_conf);
```

As shown above, queue IDs are as follows:

- id 0, atomic queue #1
- id 1, atomic queue #2
- id 2, single-link queue

These queues are used for the remainder of this walk-through.

Setting up Ports

Once queues are set up successfully, create the ports as required.

```
struct rte_event_port_conf rx_conf = {
        .dequeue_depth = 128,
        .enqueue_depth = 128,
        .new_event_threshold = 1024,
};
struct rte_event_port_conf worker_conf = {
        .dequeue_depth = 16,
        .enqueue_depth = 64,
        .new_event_threshold = 4096,
};
struct rte_event_port_conf tx_conf = {
        .dequeue_depth = 128,
```

(continues on next page)

(continued from previous page)

```
        .enqueue_depth = 128,
        .new_event_threshold = 4096,
};
int dev_id = 0;
int rx_port_id = 0;
int err = rte_event_port_setup(dev_id, rx_port_id, &rx_conf);

for(int worker_port_id = 1; worker_port_id <= 4; worker_port_id++) {
        int err = rte_event_port_setup(dev_id, worker_port_id, &worker_conf);
}

int tx_port_id = 5;
int err = rte_event_port_setup(dev_id, tx_port_id, &tx_conf);
```

As shown above:

- port 0: RX core
- ports 1,2,3,4: Workers
- port 5: TX core

These ports are used for the remainder of this walk-through.

Linking Queues and Ports

The final step is to "wire up" the ports to the queues. After this, the eventdev is capable of scheduling events, and when cores request work to do, the correct events are provided to that core. Note that the RX core takes input from e.g.: a NIC so it is not linked to any eventdev queues.

Linking all workers to atomic queues, and the TX core to the single-link queue can be achieved like this:

```
uint8_t rx_port_id = 0;
uint8_t tx_port_id = 5;
uint8_t atomic_qs[] = {0, 1};
uint8_t single_link_q = 2;
uint8_t priority = RTE_EVENT_DEV_PRIORITY_NORMAL;

for(int worker_port_id = 1; worker_port_id <= 4; worker_port_id++) {
        int links_made = rte_event_port_link(dev_id, worker_port_id, atomic_qs, NULL, 2);
}
int links_made = rte_event_port_link(dev_id, tx_port_id, &single_link_q, &priority, 1);
```

Starting the EventDev

A single function call tells the eventdev instance to start processing events. Note that all queues must be linked to for the instance to start, as if any queue is not linked to, enqueuing to that queue will cause the application to backpressure and eventually stall due to no space in the eventdev.

```
int err = rte_event_dev_start(dev_id);
```

Note: EventDev needs to be started before starting the event producers such as event_eth_rx_adapter, event_timer_adapter and event_crypto_adapter.

Ingress of New Events

Now that the eventdev is set up, and ready to receive events, the RX core must enqueue some events into the system for it to schedule. The events to be scheduled are ordinary DPDK packets, received from an eth_rx_burst() as normal. The following code shows how those packets can be enqueued into the eventdev:

```
const uint16_t nb_rx = rte_eth_rx_burst(eth_port, 0, mbufs, BATCH_SIZE);

for (i = 0; i < nb_rx; i++) {
        ev[i].flow_id = mbufs[i]->hash.rss;
        ev[i].op = RTE_EVENT_OP_NEW;
        ev[i].sched_type = RTE_SCHED_TYPE_ATOMIC;
        ev[i].queue_id = atomic_q_1;
        ev[i].event_type = RTE_EVENT_TYPE_ETHDEV;
        ev[i].sub_event_type = 0;
        ev[i].priority = RTE_EVENT_DEV_PRIORITY_NORMAL;
        ev[i].mbuf = mbufs[i];
}

const int nb_tx = rte_event_enqueue_burst(dev_id, rx_port_id, ev, nb_rx);
if (nb_tx != nb_rx) {
        for(i = nb_tx; i < nb_rx; i++)
                rte_pktmbuf_free(mbufs[i]);
}
```

Forwarding of Events

Now that the RX core has injected events, there is work to be done by the workers. Note that each worker will dequeue as many events as it can in a burst, process each one individually, and then burst the packets back into the eventdev.

The worker can lookup the events source from `event.queue_id`, which should indicate to the worker what workload needs to be performed on the event. Once done, the worker can update the `event.queue_id` to a new value, to send the event to the next stage in the pipeline.

```
int timeout = 0;
struct rte_event events[BATCH_SIZE];
uint16_t nb_rx = rte_event_dequeue_burst(dev_id, worker_port_id, events, BATCH_SIZE, timeout);

for (i = 0; i < nb_rx; i++) {
        /* process mbuf using events[i].queue_id as pipeline stage */
        struct rte_mbuf *mbuf = events[i].mbuf;
        /* Send event to next stage in pipeline */
        events[i].queue_id++;
}

uint16_t nb_tx = rte_event_enqueue_burst(dev_id, worker_port_id, events, nb_rx);
```

Egress of Events

Finally, when the packet is ready for egress or needs to be dropped, we need to inform the eventdev that the packet is no longer being handled by the application. This can be done by calling dequeue() or dequeue_burst(), which indicates that the previous burst of packets is no longer in use by the application.

An event driven worker thread has following typical workflow on fastpath:

```
while (1) {
        rte_event_dequeue_burst(...);
        (event processing)
        rte_event_enqueue_burst(...);
}
```

5.38.3 Summary

The eventdev library allows an application to easily schedule events as it requires, either using a run-to-completion or pipeline processing model. The queues and ports abstract the logical functionality of an eventdev, providing the application with a generic method to schedule events. With the flexible PMD infrastructure applications benefit of improvements in existing eventdevs and additions of new ones without modification.

5.39 Event Ethernet Rx Adapter Library

The DPDK Eventdev API allows the application to use an event driven programming model for packet processing. In this model, the application polls an event device port for receiving events that reference packets instead of polling Rx queues of ethdev ports. Packet transfer between ethdev and the event device can be supported in hardware or require a software thread to receive packets from the ethdev port using ethdev poll mode APIs and enqueue these as events to the event device using the eventdev API. Both transfer mechanisms may be present on the same platform depending on the particular combination of the ethdev and the event device. For SW based packet transfer, if the mbuf does not have a timestamp set, the adapter adds a timestamp to the mbuf using rte_get_tsc_cycles(), this provides a more accurate timestamp as compared to if the application were to set the timestamp since it avoids event device schedule latency.

The Event Ethernet Rx Adapter library is intended for the application code to configure both transfer mechanisms using a common API. A capability API allows the eventdev PMD to advertise features supported for a given ethdev and allows the application to perform configuration as per supported features.

5.39.1 API Walk-through

This section will introduce the reader to the adapter API. The application has to first instantiate an adapter which is associated with a single eventdev, next the adapter instance is configured with Rx queues that are either polled by a SW thread or linked using hardware support. Finally the adapter is started.

For SW based packet transfers from ethdev to eventdev, the adapter uses a DPDK service function and the application is also required to assign a core to the service function.

Creating an Adapter Instance

An adapter instance is created using `rte_event_eth_rx_adapter_create()`. This function is passed the event device to be associated with the adapter and port configuration for the adapter to setup an event port if the adapter needs to use a service function.

```
int err;
uint8_t dev_id;
struct rte_event_dev_info dev_info;
struct rte_event_port_conf rx_p_conf;

err = rte_event_dev_info_get(id, &dev_info);

rx_p_conf.new_event_threshold = dev_info.max_num_events;
rx_p_conf.dequeue_depth = dev_info.max_event_port_dequeue_depth;
rx_p_conf.enqueue_depth = dev_info.max_event_port_enqueue_depth;
err = rte_event_eth_rx_adapter_create(id, dev_id, &rx_p_conf);
```

If the application desires to have finer control of eventdev port allocation and setup, it can use the `rte_event_eth_rx_adapter_create_ext()` function. The `rte_event_eth_rx_adapter_create_ext()` function is passed a callback function. The callback function is invoked if the adapter needs to use a service function and needs to create an event port for it. The callback is expected to fill the `struct rte_event_eth_rx_adapter_conf structure` passed to it.

Adding Rx Queues to the Adapter Instance

Ethdev Rx queues are added to the instance using the `rte_event_eth_rx_adapter_queue_add()` function. Configuration for the Rx queue is passed in using a `struct rte_event_eth_rx_adapter_queue_conf` parameter. Event information for packets from this Rx queue is encoded in the `ev` field of `struct rte_event_eth_rx_adapter_queue_conf`. The servicing_weight member of the struct rte_event_eth_rx_adapter_queue_conf is the relative polling frequency of the Rx queue and is applicable when the adapter uses a service core function.

```
ev.queue_id = 0;
ev.sched_type = RTE_SCHED_TYPE_ATOMIC;
ev.priority = 0;

queue_config.rx_queue_flags = 0;
queue_config.ev = ev;
queue_config.servicing_weight = 1;

err = rte_event_eth_rx_adapter_queue_add(id,
                                         eth_dev_id,
                                         0, &queue_config);
```

Querying Adapter Capabilities

The `rte_event_eth_rx_adapter_caps_get()` function allows the application to query the adapter capabilities for an eventdev and ethdev combination. For e.g, if the `RTE_EVENT_ETH_RX_ADAPTER_CAP_OVERRIDE_FLOW_ID` is set, the application can override the adapter generated flow ID in the event using `rx_queue_flags` field in `struct rte_event_eth_rx_adapter_queue_conf` which is passed as a parameter to the `rte_event_eth_rx_adapter_queue_add()` function.

```
err = rte_event_eth_rx_adapter_caps_get(dev_id, eth_dev_id, &cap);

queue_config.rx_queue_flags = 0;
if (cap & RTE_EVENT_ETH_RX_ADAPTER_CAP_OVERRIDE_FLOW_ID) {
        ev.flow_id = 1;
        queue_config.rx_queue_flags =
                RTE_EVENT_ETH_RX_ADAPTER_QUEUE_FLOW_ID_VALID;
}
```

Configuring the Service Function

If the adapter uses a service function, the application is required to assign a service core to the service function as show below.

```
uint32_t service_id;

if (rte_event_eth_rx_adapter_service_id_get(0, &service_id) == 0)
        rte_service_map_lcore_set(service_id, RX_CORE_ID);
```

Starting the Adapter Instance

The application calls `rte_event_eth_rx_adapter_start()` to start the adapter. This function calls the start callbacks of the eventdev PMDs for hardware based eventdev-ethdev connections and `rte_service_run_state_set()` to enable the service function if one exists.

Note: The eventdev to which the event_eth_rx_adapter is connected needs to be started before calling rte_event_eth_rx_adapter_start().

Getting Adapter Statistics

The `rte_event_eth_rx_adapter_stats_get()` function reports counters defined in struct `rte_event_eth_rx_adapter_stats`. The received packet and enqueued event counts are a sum of the counts from the eventdev PMD callbacks if the callback is supported, and the counts maintained by the service function, if one exists. The service function also maintains a count of cycles for which it was not able to enqueue to the event device.

Interrupt Based Rx Queues

The service core function is typically set up to poll ethernet Rx queues for packets. Certain queues may have low packet rates and it would be more efficient to enable the Rx queue interrupt and read packets after receiving the interrupt.

The servicing_weight member of struct rte_event_eth_rx_adapter_queue_conf is applicable when the adapter uses a service core function. The application has to enable Rx queue interrupts when configuring the ethernet device using the `rte_eth_dev_configure()` function and then use a servicing_weight of zero when adding the Rx queue to the adapter.

The adapter creates a thread blocked on the interrupt, on an interrupt this thread enqueues the port id and the queue id to a ring buffer. The adapter service function dequeues the port id and queue id from the ring buffer, invokes the `rte_eth_rx_burst()` to receive packets on the queue and converts the received packets to events in the same manner as packets received on a polled Rx queue. The interrupt thread is affinitized to the same CPUs as the lcores of the Rx adapter service function, if the Rx adapter service function has not been mapped to any lcores, the interrupt thread is mapped to the master lcore.

Rx Callback for SW Rx Adapter

For SW based packet transfers, i.e., when the `RTE_EVENT_ETH_RX_ADAPTER_CAP_INTERNAL_PORT` is not set in the adapter's capabilities flags for a particular ethernet device, the service function temporarily enqueues mbufs to an event buffer before batch enqueuing these to the event device. If the buffer fills up, the service function stops dequeuing packets from the ethernet device. The application may want to monitor the buffer fill level and instruct the service function to selectively enqueue packets to the event device. The application may also use some other criteria to decide which packets should enter the event device even when the event buffer fill level is low. The `rte_event_eth_rx_adapter_cb_register()` function allow the application to register a callback that selects which packets to enqueue to the event device.

5.40 Event Ethernet Tx Adapter Library

The DPDK Eventdev API allows the application to use an event driven programming model for packet processing in which the event device distributes events referencing packets to the application cores in a dynamic load balanced fashion while handling atomicity and packet ordering. Event adapters provide the interface between the ethernet, crypto and timer devices and the event device. Event adapter APIs enable common application code by abstracting PMD specific capabilities. The Event ethernet Tx adapter provides configuration and data path APIs for the transmit stage of the application allowing the same application code to use eventdev PMD support or in its absence, a common implementation.

In the common implementation, the application enqueues mbufs to the adapter which runs as a rte_service function. The service function dequeues events from its event port and transmits the mbufs referenced by these events.

5.40.1 API Walk-through

This section will introduce the reader to the adapter API. The application has to first instantiate an adapter which is associated with a single eventdev, next the adapter instance is configured with Tx queues, finally the adapter is started and the application can start enqueuing mbufs to it.

Creating an Adapter Instance

An adapter instance is created using `rte_event_eth_tx_adapter_create()`. This function is passed the event device to be associated with the adapter and port configuration for the adapter to setup an event port if the adapter needs to use a service function.

If the application desires to have finer control of eventdev port configuration, it can use the `rte_event_eth_tx_adapter_create_ext()` function. The `rte_event_eth_tx_adapter_create_ext()` function is passed a callback function. The callback function is invoked if the adapter needs to use a service function and needs to create an event port for it. The callback is expected to fill the `struct rte_event_eth_tx_adapter_conf` structure passed to it.

```
struct rte_event_dev_info dev_info;
struct rte_event_port_conf tx_p_conf = {0};

err = rte_event_dev_info_get(id, &dev_info);

tx_p_conf.new_event_threshold = dev_info.max_num_events;
tx_p_conf.dequeue_depth = dev_info.max_event_port_dequeue_depth;
tx_p_conf.enqueue_depth = dev_info.max_event_port_enqueue_depth;

err = rte_event_eth_tx_adapter_create(id, dev_id, &tx_p_conf);
```

Adding Tx Queues to the Adapter Instance

Ethdev Tx queues are added to the instance using the `rte_event_eth_tx_adapter_queue_add()` function. A queue value of -1 is used to indicate all queues within a device.

```
int err = rte_event_eth_tx_adapter_queue_add(id,
                                             eth_dev_id,
                                             q);
```

Querying Adapter Capabilities

The `rte_event_eth_tx_adapter_caps_get()` function allows the application to query the adapter capabilities for an eventdev and ethdev combination. Currently, the only capability flag defined is `RTE_EVENT_ETH_TX_ADAPTER_CAP_INTERNAL_PORT`, the application can query this flag to determine if a service function is associated with the adapter and retrieve its service identifier using the `rte_event_eth_tx_adapter_service_id_get()` API.

```
int err = rte_event_eth_tx_adapter_caps_get(dev_id, eth_dev_id, &cap);

if (!(cap & RTE_EVENT_ETH_TX_ADAPTER_CAP_INTERNAL_PORT))
        err = rte_event_eth_tx_adapter_service_id_get(id, &service_id);
```

Linking a Queue to the Adapter's Event Port

If the adapter uses a service function as described in the previous section, the application is required to link a queue to the adapter's event port. The adapter's event port can be obtained using the `rte_event_eth_tx_adapter_event_port_get()` function. The queue can be configured with the `RTE_EVENT_QUEUE_CFG_SINGLE_LINK` since it is linked to a single event port.

Configuring the Service Function

If the adapter uses a service function, the application can assign a service core to the service function as shown below.

```
if (rte_event_eth_tx_adapter_service_id_get(id, &service_id) == 0)
        rte_service_map_lcore_set(service_id, TX_CORE_ID);
```

Starting the Adapter Instance

The application calls `rte_event_eth_tx_adapter_start()` to start the adapter. This function calls the start callback of the eventdev PMD if supported, and the `rte_service_run_state_set()` to enable the service function if one exists.

Enqueuing Packets to the Adapter

The application needs to notify the adapter about the transmit port and queue used to send the packet. The transmit port is set in the `struct rte mbuf::port` field and the transmit queue is set using the `rte_event_eth_tx_adapter_txq_set()` function.

If the eventdev PMD supports the `RTE_EVENT_ETH_TX_ADAPTER_CAP_INTERNAL_PORT` capability for a given ethernet device, the application should use the `rte_event_eth_tx_adapter_enqueue()` function to enqueue packets to the adapter.

If the adapter uses a service function for the ethernet device then the application should use the `rte_event_enqueue_burst()` function.

```
struct rte_event event;

if (cap & RTE_EVENT_ETH_TX_ADAPTER_CAP_INTERNAL_PORT) {

        event.mbuf = m;
        eq_flags = 0;

        m->port = tx_port;
        rte_event_eth_tx_adapter_txq_set(m, tx_queue_id);

        rte_event_eth_tx_adapter_enqueue(dev_id, ev_port, &event, 1, eq_flags);
} else {

        event.queue_id = qid; /* event queue linked to adapter port */
        event.op = RTE_EVENT_OP_NEW;
        event.event_type = RTE_EVENT_TYPE_CPU;
        event.sched_type = RTE_SCHED_TYPE_ATOMIC;
        event.mbuf = m;

        m->port = tx_port;
```

(continues on next page)

(continued from previous page)

```
        rte_event_eth_tx_adapter_txq_set(m, tx_queue_id);

        rte_event_enqueue_burst(dev_id, ev_port, &event, 1);
}
```

Getting Adapter Statistics

The `rte_event_eth_tx_adapter_stats_get()` function reports counters defined in struct `rte_event_eth_tx_adapter_stats`. The counter values are the sum of the counts from the eventdev PMD callback if the callback is supported, and the counts maintained by the service function, if one exists.

5.41 Event Timer Adapter Library

The DPDK *Event Device library* introduces an event driven programming model which presents applications with an alternative to the polling model traditionally used in DPDK applications. Event devices can be coupled with arbitrary components to provide new event sources by using **event adapters**. The Event Timer Adapter is one such adapter; it bridges event devices and timer mechanisms.

The Event Timer Adapter library extends the event driven model by introducing a *new type of event* that represents a timer expiration, and providing an API with which adapters can be created or destroyed, and *event timers* can be armed and canceled.

The Event Timer Adapter library is designed to interface with hardware or software implementations of the timer mechanism; it will query an eventdev PMD to determine which implementation should be used. The default software implementation manages timers using the DPDK *Timer library*.

Examples of using the API are presented in the *API Overview* and *Processing Timer Expiry Events* sections. Code samples are abstracted and are based on the example of handling a TCP retransmission.

5.41.1 Event Timer struct

Event timers are timers that enqueue a timer expiration event to an event device upon timer expiration.

The Event Timer Adapter API represents each event timer with a generic struct, which contains an event and user metadata. The `rte_event_timer` struct is defined in `lib/librte_event/librte_event_timer_adapter.h`.

Timer Expiry Event

The event contained by an event timer is enqueued in the event device when the timer expires, and the event device uses the attributes below when scheduling it:

- `event_queue_id` - Application should set this to specify an event queue to which the timer expiry event should be enqueued
- `event_priority` - Application can set this to indicate the priority of the timer expiry event in the event queue relative to other events
- `sched_type` - Application can set this to specify the scheduling type of the timer expiry event

- `flow_id` - Application can set this to indicate which flow this timer expiry event corresponds to
- `op` - Will be set to `RTE_EVENT_OP_NEW` by the event timer adapter
- `event_type` - Will be set to `RTE_EVENT_TYPE_TIMER` by the event timer adapter

Timeout Ticks

The number of ticks from now in which the timer will expire. The ticks value has a resolution (`timer_tick_ns`) that is specified in the event timer adapter configuration.

State

Before arming an event timer, the application should initialize its state to RTE_EVENT_TIMER_NOT_ARMED. The event timer's state will be updated when a request to arm or cancel it takes effect.

If the application wishes to rearm the timer after it has expired, it should reset the state back to RTE_EVENT_TIMER_NOT_ARMED before doing so.

User Metadata

Memory to store user specific metadata. The event timer adapter implementation will not modify this area.

5.41.2 API Overview

This section will introduce the reader to the event timer adapter API, showing how to create and configure an event timer adapter and use it to manage event timers.

From a high level, the setup steps are:

- rte_event_timer_adapter_create()
- rte_event_timer_adapter_start()

And to start and stop timers:

- rte_event_timer_arm_burst()
- rte_event_timer_cancel_burst()

Create and Configure an Adapter Instance

To create an event timer adapter instance, initialize an `rte_event_timer_adapter_conf` struct with the desired values, and pass it to `rte_event_timer_adapter_create()`.

```
#define NSECPERSEC 1E9 // No of ns in 1 sec
const struct rte_event_timer_adapter_conf adapter_config = {
        .event_dev_id = event_dev_id,
        .timer_adapter_id = 0,
        .clk_src = RTE_EVENT_TIMER_ADAPTER_CPU_CLK,
        .timer_tick_ns = NSECPERSEC / 10, // 100 milliseconds
```

(continues on next page)

(continued from previous page)

```
        .max_tmo_nsec = 180 * NSECPERSEC // 2 minutes
        .nb_timers = 40000,
        .timer_adapter_flags = 0,
};

struct rte_event_timer_adapter *adapter = NULL;
adapter = rte_event_timer_adapter_create(&adapter_config);

if (adapter == NULL) { ... };
```

Before creating an instance of a timer adapter, the application should create and configure an event device along with its event ports. Based on the event device capability, it might require creating an additional event port to be used by the timer adapter. If required, the `rte_event_timer_adapter_create()` function will use a default method to configure an event port; it will examine the current event device configuration, determine the next available port identifier number, and create a new event port with a default port configuration.

If the application desires to have finer control of event port allocation and setup, it can use the `rte_event_timer_adapter_create_ext()` function. This function is passed a callback function that will be invoked if the adapter needs to create an event port, giving the application the opportunity to control how it is done.

Retrieve Event Timer Adapter Contextual Information

The event timer adapter implementation may have constraints on tick resolution or maximum timer expiry timeout based on the given event timer adapter or system. In this case, the implementation may adjust the tick resolution or maximum timeout to the best possible configuration.

Upon successful event timer adapter creation, the application can get the configured resolution and max timeout with `rte_event_timer_adapter_get_info()`. This function will return an `rte_event_timer_adapter_info` struct, which contains the following members:

- `min_resolution_ns` - Minimum timer adapter tick resolution in ns.
- `max_tmo_ns` - Maximum timer timeout(expiry) in ns.
- `adapter_conf` - Configured event timer adapter attributes

Configuring the Service Component

If the adapter uses a service component, the application is required to map the service to a service core before starting the adapter:

```
uint32_t service_id;

if (rte_event_timer_adapter_service_id_get(adapter, &service_id) == 0)
        rte_service_map_lcore_set(service_id, EVTIM_CORE_ID);
```

An event timer adapter uses a service component if the event device PMD indicates that the adapter should use a software implementation.

Starting the Adapter Instance

The application should call `rte_event_timer_adapter_start()` to start running the event timer adapter. This function calls the start entry points defined by eventdev PMDs for hardware implementations or puts a service component into the running state in the software implementation.

Note: The eventdev to which the event_timer_adapter is connected needs to be started before calling rte_event_timer_adapter_start().

Arming Event Timers

Once an event timer adapter has been started, an application can begin to manage event timers with it.

The application should allocate `struct rte_event_timer` objects from a mempool or huge-page backed application buffers of required size. Upon successful allocation, the application should initialize the event timer, and then set any of the necessary event attributes described in the *Timer Expiry Event* section. In the following example, assume `conn` represents a TCP connection and that `event_timer_pool` is a mempool that was created previously:

```
rte_mempool_get(event_timer_pool, (void **)&conn->evtim);
if (conn->evtim == NULL) { ... }

/* Set up the event timer. */
conn->evtim->ev.op = RTE_EVENT_OP_NEW;
conn->evtim->ev.queue_id = event_queue_id;
conn->evtim->ev.sched_type = RTE_SCHED_TYPE_ATOMIC;
conn->evtim->ev.priority = RTE_EVENT_DEV_PRIORITY_NORMAL;
conn->evtim->ev.event_type = RTE_EVENT_TYPE_TIMER;
conn->evtim->ev.event_ptr = conn;
conn->evtim->state = RTE_EVENT_TIMER_NOT_ARMED;
conn->evtim->timeout_ticks = 30; //3 sec Per RFC1122(TCP returns)
```

Note that it is necessary to initialize the event timer state to RTE_EVENT_TIMER_NOT_ARMED. Also note that we have saved a pointer to the `conn` object in the timer's event payload. This will allow us to locate the connection object again once we dequeue the timer expiry event from the event device later. As a convenience, the application may specify no value for ev.event_ptr, and the adapter will by default set it to point at the event timer itself.

Now we can arm the event timer with `rte_event_timer_arm_burst()`:

```
ret = rte_event_timer_arm_burst(adapter, &conn->evtim, 1);
if (ret != 1) { ... }
```

Once an event timer expires, the application may free it or rearm it as necessary. If the application will rearm the timer, the state should be reset to RTE_EVENT_TIMER_NOT_ARMED by the application before rearming it.

Multiple Event Timers with Same Expiry Value

In the special case that there is a set of event timers that should all expire at the same time, the application may call `rte_event_timer_arm_tmo_tick_burst()`, which allows the implementation to optimize the operation if possible.

Canceling Event Timers

An event timer that has been armed as described in *Arming Event Timers* can be canceled by calling `rte_event_timer_cancel_burst()`:

```
/* Ack for the previous tcp data packet has been received;
 * cancel the retransmission timer
 */
rte_event_timer_cancel_burst(adapter, &conn->timer, 1);
```

5.41.3 Processing Timer Expiry Events

Once an event timer has successfully enqueued a timer expiry event in the event device, the application will subsequently dequeue it from the event device. The application can use the event payload to retrieve a pointer to the object associated with the event timer. It can then re-arm the event timer or free the event timer object as desired:

```
void
event_processing_loop(...)
{
        while (...) {
                /* Receive events from the configured event port. */
                rte_event_dequeue_burst(event_dev_id, event_port, &ev, 1, 0);
                ...
                switch(ev.event_type) {
                        ...
                        case RTE_EVENT_TYPE_TIMER:
                                process_timer_event(ev);
                                ...
                                break;
                }
        }
}

uint8_t
process_timer_event(...)
{
        /* A retransmission timeout for the connection has been received. */
        conn = ev.event_ptr;
        /* Retransmit last packet (e.g. TCP segment). */
        ...
        /* Re-arm timer using original values. */
        rte_event_timer_arm_burst(adapter_id, &conn->timer, 1);
}
```

5.41.4 Summary

The Event Timer Adapter library extends the DPDK event-based programming model by representing timer expirations as events in the system and allowing applications to use existing event processing loops to arm and cancel event timers or handle timer expiry events.

5.42 Event Crypto Adapter Library

The DPDK *Eventdev library* provides event driven programming model with features to schedule events. The *Cryptodev library* provides an interface to the crypto poll mode drivers which supports different crypto operations. The Event Crypto Adapter is one of the adapter which is intended to bridge between the event device and the crypto device.

The packet flow from crypto device to the event device can be accomplished using SW and HW based transfer mechanism. The Adapter queries an eventdev PMD to determine which mechanism to be used. The adapter uses an EAL service core function for SW based packet transfer and uses the eventdev PMD functions to configure HW based packet transfer between the crypto device and the event device. The crypto adapter uses a new event type called `RTE_EVENT_TYPE_CRYPTODEV` to indicate the event source.

The application can choose to submit a crypto operation directly to crypto device or send it to the crypto adapter via eventdev based on RTE_EVENT_CRYPTO_ADAPTER_CAP_INTERNAL_PORT_OP_FWD capability. The first mode is known as the event new(RTE_EVENT_CRYPTO_ADAPTER_OP_NEW) mode and the second as the event forward(RTE_EVENT_CRYPTO_ADAPTER_OP_FORWARD) mode. The choice of mode can be specified while creating the adapter. In the former mode, it is an application responsibility to enable ingress packet ordering. In the latter mode, it is the adapter responsibility to enable the ingress packet ordering.

5.42.1 Adapter Mode

RTE_EVENT_CRYPTO_ADAPTER_OP_NEW mode

In the RTE_EVENT_CRYPTO_ADAPTER_OP_NEW mode, application submits crypto operations directly to crypto device. The adapter then dequeues crypto completions from crypto device and enqueues them as events to the event device. This mode does not ensure ingress ordering, if the application directly enqueues to the cryptodev without going through crypto/atomic stage. In this mode, events dequeued from the adapter will be treated as new events. The application needs to specify event information (response information) which is needed to enqueue an event after the crypto operation is completed.

Fig. 5.66: Working model of `RTE_EVENT_CRYPTO_ADAPTER_OP_NEW` mode

RTE_EVENT_CRYPTO_ADAPTER_OP_FORWARD mode

In the RTE_EVENT_CRYPTO_ADAPTER_OP_FORWARD mode, if HW supports RTE_EVENT_CRYPTO_ADAPTER_CAP_INTERNAL_PORT_OP_FWD capability the application can directly submit the crypto operations to the cryptodev. If not, application retrieves crypto adapter's event port using rte_event_crypto_adapter_event_port_get() API. Then, links its event queue to this port and starts enqueuing crypto operations as events to the eventdev. The adapter then dequeues the events and submits the crypto operations to the cryptodev. After the crypto completions, the adapter enqueues events to the event device. Application can use this mode, when ingress packet ordering is needed. In this mode, events dequeued from the adapter will be treated as forwarded events. The application needs to specify the cryptodev ID and queue pair ID (request information) needed to enqueue a crypto operation in addition to the event information (response information) needed to enqueue an event after the crypto operation has completed.

Fig. 5.67: Working model of RTE_EVENT_CRYPTO_ADAPTER_OP_FORWARD mode

5.42.2 API Overview

This section has a brief introduction to the event crypto adapter APIs. The application is expected to create an adapter which is associated with a single eventdev, then add cryptodev and queue pair to the adapter instance.

Create an adapter instance

An adapter instance is created using `rte_event_crypto_adapter_create()`. This function is called with event device to be associated with the adapter and port configuration for the adapter to setup an event port(if the adapter needs to use a service function).

Adapter can be started in RTE_EVENT_CRYPTO_ADAPTER_OP_NEW or RTE_EVENT_CRYPTO_ADAPTER_OP_FORWARD mode.

```
int err;
uint8_t dev_id, id;
struct rte_event_dev_info dev_info;
struct rte_event_port_conf conf;
enum rte_event_crypto_adapter_mode mode;

err = rte_event_dev_info_get(id, &dev_info);

conf.new_event_threshold = dev_info.max_num_events;
conf.dequeue_depth = dev_info.max_event_port_dequeue_depth;
conf.enqueue_depth = dev_info.max_event_port_enqueue_depth;
mode = RTE_EVENT_CRYPTO_ADAPTER_OP_FORWARD;
err = rte_event_crypto_adapter_create(id, dev_id, &conf, mode);
```

If the application desires to have finer control of eventdev port allocation and setup, it can use the `rte_event_crypto_adapter_create_ext()` function. The `rte_event_crypto_adapter_create_ext()` function is passed as a callback function. The callback function is invoked if the adapter needs to use a service function and needs to create an event port for it. The callback is expected to fill the `struct rte_event_crypto_adapter_conf` structure passed to it.

For RTE_EVENT_CRYPTO_ADAPTER_OP_FORWARD mode, the event port created by adapter can be retrieved using `rte_event_crypto_adapter_event_port_get()` API. Application can use this event port to link with event queue on which it enqueues events towards the crypto adapter.

```
uint8_t id, evdev, crypto_ev_port_id, app_qid;
struct rte_event ev;
int ret;

ret = rte_event_crypto_adapter_event_port_get(id, &crypto_ev_port_id);
ret = rte_event_queue_setup(evdev, app_qid, NULL);
ret = rte_event_port_link(evdev, crypto_ev_port_id, &app_qid, NULL, 1);

// Fill in event info and update event_ptr with rte_crypto_op
memset(&ev, 0, sizeof(ev));
ev.queue_id = app_qid;
.
.
ev.event_ptr = op;
ret = rte_event_enqueue_burst(evdev, app_ev_port_id, ev, nb_events);
```

Querying adapter capabilities

The `rte_event_crypto_adapter_caps_get()` function allows the application to query the adapter capabilities for an eventdev and cryptodev combination. This API provides whether cryptodev and eventdev are connected using internal HW port or not.

```
rte_event_crypto_adapter_caps_get(dev_id, cdev_id, &cap);
```

Adding queue pair to the adapter instance

Cryptodev device id and queue pair are created using cryptodev APIs. For more information see *here*.

```
struct rte_cryptodev_config conf;
struct rte_cryptodev_qp_conf qp_conf;
uint8_t cdev_id = 0;
uint16_t qp_id = 0;

rte_cryptodev_configure(cdev_id, &conf);
rte_cryptodev_queue_pair_setup(cdev_id, qp_id, &qp_conf);
```

These cryptodev id and queue pair are added to the instance using the `rte_event_crypto_adapter_queue_pair_add()` API. The same is removed using `rte_event_crypto_adapter_queue_pair_del()` API. If HW supports RTE_EVENT_CRYPTO_ADAPTER_CAP_INTERNAL_PORT_QP_EV_BIND capability, event information must be passed to the add API.

```
uint32_t cap;
int ret;

ret = rte_event_crypto_adapter_caps_get(id, evdev, &cap);
if (cap & RTE_EVENT_CRYPTO_ADAPTER_CAP_INTERNAL_PORT_QP_EV_BIND) {
        struct rte_event event;

        // Fill in event information & pass it to add API
        rte_event_crypto_adapter_queue_pair_add(id, cdev_id, qp_id, &event);
```

(continues on next page)

(continued from previous page)

```
} else
        rte_event_crypto_adapter_queue_pair_add(id, cdev_id, qp_id, NULL);
```

Configure the service function

If the adapter uses a service function, the application is required to assign a service core to the service function as show below.

```
uint32_t service_id;

if (rte_event_crypto_adapter_service_id_get(id, &service_id) == 0)
        rte_service_map_lcore_set(service_id, CORE_ID);
```

Set event request/response information

In the RTE_EVENT_CRYPTO_ADAPTER_OP_FORWARD mode, the application needs to specify the cryptodev ID and queue pair ID (request information) in addition to the event information (response information) needed to enqueue an event after the crypto operation has completed. The request and response information are specified in the `struct rte_crypto_op` private data or session's private data.

In the RTE_EVENT_CRYPTO_ADAPTER_OP_NEW mode, the application is required to provide only the response information.

The SW adapter or HW PMD uses `rte_crypto_op::sess_type` to decide whether request/response data is located in the crypto session/ crypto security session or at an offset in the `struct rte_crypto_op`. The `rte_crypto_op::private_data_offset` is used to locate the request/ response in the `rte_crypto_op`.

For crypto session, `rte_cryptodev_sym_session_set_user_data()` API will be used to set request/response data. The same data will be obtained by `rte_cryptodev_sym_session_get_user_data()` API. The RTE_EVENT_CRYPTO_ADAPTER_CAP_SESSION_PRIVATE_DATA capability indicates whether HW or SW supports this feature.

For security session, `rte_security_session_set_private_data()` API will be used to set request/response data. The same data will be obtained by `rte_security_session_get_private_data()` API.

For session-less it is mandatory to place the request/response data with the `rte_crypto_op`.

```
union rte_event_crypto_metadata m_data;
struct rte_event ev;
struct rte_crypto_op *op;

/* Allocate & fill op structure */
op = rte_crypto_op_alloc();

memset(&m_data, 0, sizeof(m_data));
memset(&ev, 0, sizeof(ev));
/* Fill event information and update event_ptr to rte_crypto_op */
ev.event_ptr = op;

if (op->sess_type == RTE_CRYPTO_OP_WITH_SESSION) {
        /* Copy response information */
```

(continues on next page)

(continued from previous page)

```
        rte_memcpy(&m_data.response_info, &ev, sizeof(ev));
        /* Copy request information */
        m_data.request_info.cdev_id = cdev_id;
        m_data.request_info.queue_pair_id = qp_id;
        /* Call set API to store private data information */
        rte_cryptodev_sym_session_set_user_data(
                op->sym->session,
                &m_data,
                sizeof(m_data));
} if (op->sess_type == RTE_CRYPTO_OP_SESSIONLESS) {
        uint32_t len = IV_OFFSET + MAXIMUM_IV_LENGTH +
                        (sizeof(struct rte_crypto_sym_xform) * 2);
        op->private_data_offset = len;
        /* Copy response information */
        rte_memcpy(&m_data.response_info, &ev, sizeof(ev));
        /* Copy request information */
        m_data.request_info.cdev_id = cdev_id;
        m_data.request_info.queue_pair_id = qp_id;
        /* Store private data information along with rte_crypto_op */
        rte_memcpy(op + len, &m_data, sizeof(m_data));
}
```

Start the adapter instance

The application calls `rte_event_crypto_adapter_start()` to start the adapter. This function calls the start callbacks of the eventdev PMDs for hardware based eventdev-cryptodev connections and `rte_service_run_state_set()` to enable the service function if one exists.

```
rte_event_crypto_adapter_start(id, mode);
```

Note: The eventdev to which the event_crypto_adapter is connected needs to be started before calling rte_event_crypto_adapter_start().

Get adapter statistics

The `rte_event_crypto_adapter_stats_get()` function reports counters defined in struct `rte_event_crypto_adapter_stats`. The received packet and enqueued event counts are a sum of the counts from the eventdev PMD callbacks if the callback is supported, and the counts maintained by the service function, if one exists.

5.43 Quality of Service (QoS) Framework

This chapter describes the DPDK Quality of Service (QoS) framework.

5.43.1 Packet Pipeline with QoS Support

An example of a complex packet processing pipeline with QoS support is shown in the following figure.

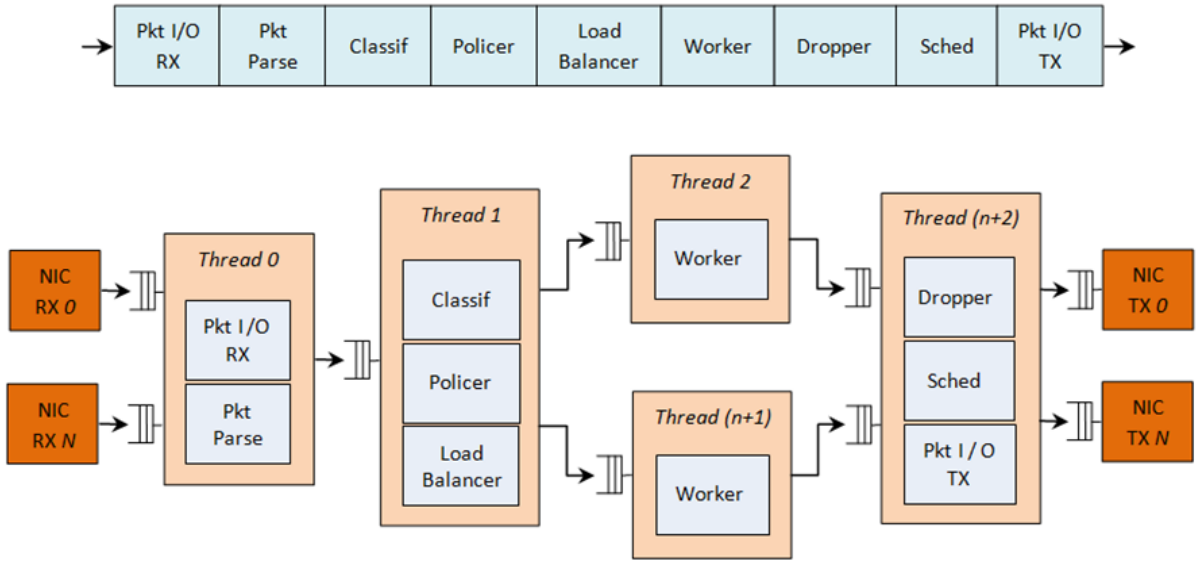


Fig. 5.68: Complex Packet Processing Pipeline with QoS Support

This pipeline can be built using reusable DPDK software libraries. The main blocks implementing QoS in this pipeline are: the policer, the dropper and the scheduler. A functional description of each block is provided in the following table.

Table 5.96: Packet Processing Pipeline Implementing QoS

#	Block	Functional Description
1	Packet I/O RX & TX	Packet reception/ transmission from/to multiple NIC ports. Poll mode drivers (PMDs) for Intel 1 GbE/10 GbE NICs.
2	Packet parser	Identify the protocol stack of the input packet. Check the integrity of the packet headers.
3	Flow classification	Map the input packet to one of the known traffic flows. Exact match table lookup using configurable hash function (jhash, CRC and so on) and bucket logic to handle collisions.
4	Policer	Packet metering using srTCM (RFC 2697) or trTCM (RFC2698) algorithms.
5	Load Balancer	Distribute the input packets to the application workers. Provide uniform load to each worker. Preserve the affinity of traffic flows to workers and the packet order within each flow.
6	Worker threads	Placeholders for the customer specific application workload (for example, IP stack and so on).
7	Dropper	Congestion management using the Random Early Detection (RED) algorithm (specified by the Sally Floyd - Van Jacobson paper) or Weighted RED (WRED). Drop packets based on the current scheduler queue load level and packet priority. When congestion is experienced, lower priority packets are dropped first.
8	Hierarchical Scheduler	5-level hierarchical scheduler (levels are: output port, subport, pipe, traffic class and queue) with thousands (typically 64K) leaf nodes (queues). Implements traffic shaping (for subport and pipe levels), strict priority (for traffic class level) and Weighted Round Robin (WRR) (for queues within each pipe traffic class).

The infrastructure blocks used throughout the packet processing pipeline are listed in the following table.

Table 5.97: Infrastructure Blocks Used by the Packet Processing Pipeline

#	Block	Functional Description
1	Buffer manager	Support for global buffer pools and private per-thread buffer caches.
2	Queue manager	Support for message passing between pipeline blocks.
3	Power saving	Support for power saving during low activity periods.

The mapping of pipeline blocks to CPU cores is configurable based on the performance level required by each specific application and the set of features enabled for each block. Some blocks might consume more than one CPU core (with each CPU core running a different instance of the same block on different input packets), while several other blocks could be mapped to the same CPU core.

5.43.2 Hierarchical Scheduler

The hierarchical scheduler block, when present, usually sits on the TX side just before the transmission stage. Its purpose is to prioritize the transmission of packets from different users and different traffic classes according to the policy specified by the Service Level Agreements (SLAs) of each network node.

Overview

The hierarchical scheduler block is similar to the traffic manager block used by network processors that typically implement per flow (or per group of flows) packet queuing and scheduling. It typically acts like a buffer that is able to temporarily store a large number of packets just before their transmission (enqueue operation); as the NIC TX is requesting more packets for transmission, these packets are later on removed and handed over to the NIC TX with the packet selection logic observing the predefined SLAs (dequeue operation).

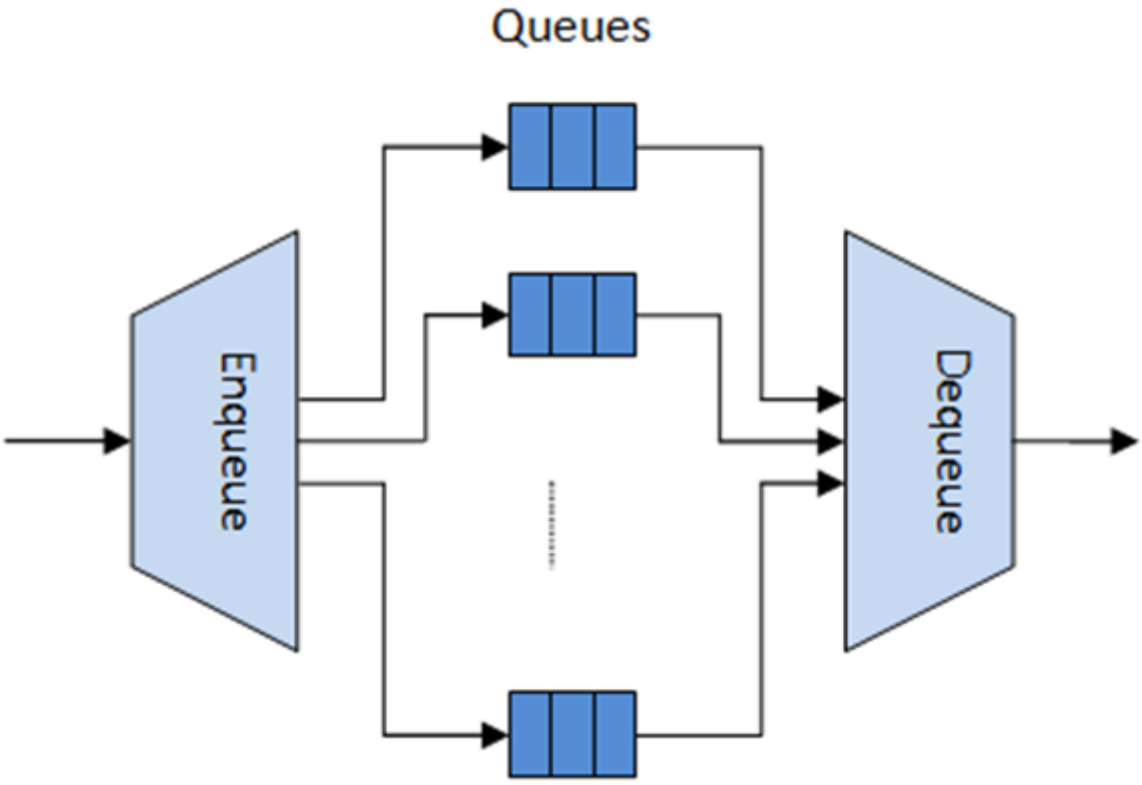


Fig. 5.69: Hierarchical Scheduler Block Internal Diagram

The hierarchical scheduler is optimized for a large number of packet queues. When only a small number of queues are needed, message passing queues should be used instead of this block. See *Worst Case Scenarios for Performance* for a more detailed discussion.

Scheduling Hierarchy

The scheduling hierarchy is shown in Fig. 5.70. The first level of the hierarchy is the Ethernet TX port 1/10/40 GbE, with subsequent hierarchy levels defined as subport, pipe, traffic class and queue.

Typically, each subport represents a predefined group of users, while each pipe represents an individual user/subscriber. Each traffic class is the representation of a different traffic type with specific loss rate, delay and jitter requirements, such as voice, video or data transfers. Each queue hosts packets from one or multiple connections of the same type belonging to the same user.

Fig. 5.70: Scheduling Hierarchy per Port

The functionality of each hierarchical level is detailed in the following table.

Table 5.98: Port Scheduling Hierarchy

#	Level	Siblings per Parent	Functional Description
1	Port	•	1. Output Ethernet port 1/10/40 GbE. 2. Multiple ports are scheduled in round robin order with all ports having equal priority.
2	Subport	Configurable (default: 8)	1. Traffic shaping using token bucket algorithm (one token bucket per subport). 2. Upper limit enforced per Traffic Class (TC) at the subport level. 3. Lower priority TCs able to reuse subport bandwidth currently unused by higher priority TCs.
3	Pipe	Configurable (default: 4K)	1. Traffic shaping using the token bucket algorithm (one token bucket per pipe.
4	Traffic Class (TC)	13	1. TCs of the same pipe handled in strict priority order. 2. Upper limit enforced per TC at the pipe level. 3. Lower priority TCs able to reuse pipe bandwidth currently unused by higher priority TCs. 4. When subport

Application Programming Interface (API)

Port Scheduler Configuration API

The rte_sched.h file contains configuration functions for port, subport and pipe.

Port Scheduler Enqueue API

The port scheduler enqueue API is very similar to the API of the DPDK PMD TX function.

```
int rte_sched_port_enqueue(struct rte_sched_port *port, struct rte_mbuf **pkts, uint32_t n_
↪pkts);
```

Port Scheduler Dequeue API

The port scheduler dequeue API is very similar to the API of the DPDK PMD RX function.

```
int rte_sched_port_dequeue(struct rte_sched_port *port, struct rte_mbuf **pkts, uint32_t n_
↪pkts);
```

Usage Example

```
/* File "application.c" */

#define N_PKTS_RX   64
#define N_PKTS_TX   48
#define NIC_RX_PORT 0
#define NIC_RX_QUEUE 0
#define NIC_TX_PORT 1
#define NIC_TX_QUEUE 0

struct rte_sched_port *port = NULL;
struct rte_mbuf *pkts_rx[N_PKTS_RX], *pkts_tx[N_PKTS_TX];
uint32_t n_pkts_rx, n_pkts_tx;

/* Initialization */

<initialization code>

/* Runtime */
while (1) {
    /* Read packets from NIC RX queue */

    n_pkts_rx = rte_eth_rx_burst(NIC_RX_PORT, NIC_RX_QUEUE, pkts_rx, N_PKTS_RX);

    /* Hierarchical scheduler enqueue */

    rte_sched_port_enqueue(port, pkts_rx, n_pkts_rx);

    /* Hierarchical scheduler dequeue */

    n_pkts_tx = rte_sched_port_dequeue(port, pkts_tx, N_PKTS_TX);
```

(continues on next page)

(continued from previous page)

```
    /* Write packets to NIC TX queue */

    rte_eth_tx_burst(NIC_TX_PORT, NIC_TX_QUEUE, pkts_tx, n_pkts_tx);
}
```

Implementation

Internal Data Structures per Port

A schematic of the internal data structures in shown in with details in.

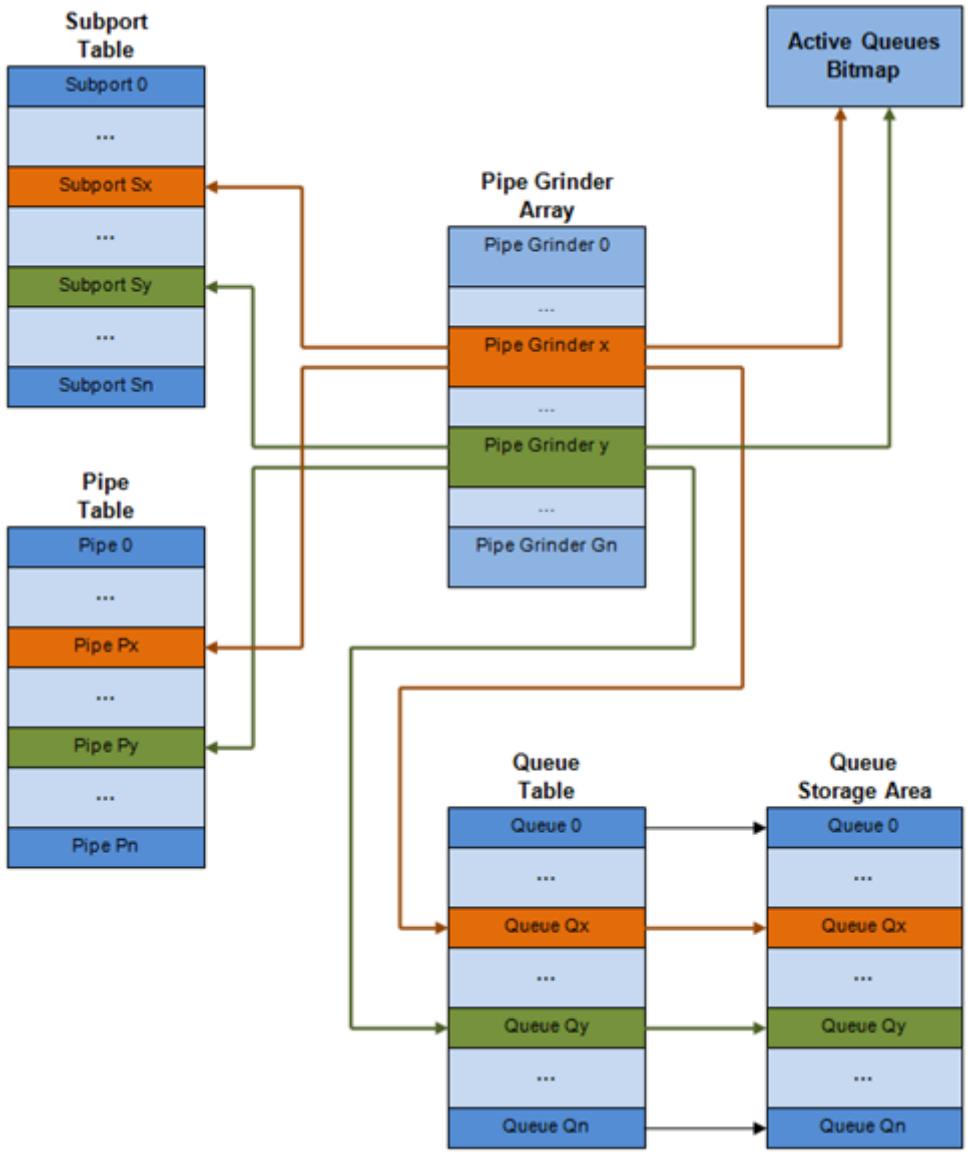


Fig. 5.71: Internal Data Structures per Port

Table 5.99: Scheduler Internal Data Structures per Port

#	Data structure	Size (bytes)	# per port	Access type		Description
				Enq	Deq	
1	Subport table entry	64	# subports per port	•	Rd, Wr	Persistent subport data (credits, etc).
2	Pipe table entry	64	# pipes per port	•	Rd, Wr	Persistent data for pipe, its TCs and its queues (credits, etc) that is updated during run-time. The pipe configuration parameters do not change during run-time. The same pipe configuration parameters are shared by multiple pipes, therefore they are not part of pipe table entry.
3	Queue table entry	4	#queues per port	Rd, Wr	Rd, Wr	Persistent queue data (read and write pointers). The queue size is the same per TC for all queues, allowing the queue base address to be computed using a fast formula, so these two parameters are not part

Multicore Scaling Strategy

The multicore scaling strategy is:

1. Running different physical ports on different threads. The enqueue and dequeue of the same port are run by the same thread.
2. Splitting the same physical port to different threads by running different sets of subports of the same physical port (virtual ports) on different threads. Similarly, a subport can be split into multiple subports that are each run by a different thread. The enqueue and dequeue of the same port are run by the same thread. This is only required if, for performance reasons, it is not possible to handle a full port with a single core.

Enqueue and Dequeue for the Same Output Port

Running enqueue and dequeue operations for the same output port from different cores is likely to cause significant impact on scheduler's performance and it is therefore not recommended.

The port enqueue and dequeue operations share access to the following data structures:

1. Packet descriptors
2. Queue table
3. Queue storage area
4. Bitmap of active queues

The expected drop in performance is due to:

1. Need to make the queue and bitmap operations thread safe, which requires either using locking primitives for access serialization (for example, spinlocks/ semaphores) or using atomic primitives for lockless access (for example, Test and Set, Compare And Swap, an so on). The impact is much higher in the former case.
2. Ping-pong of cache lines storing the shared data structures between the cache hierarchies of the two cores (done transparently by the MESI protocol cache coherency CPU hardware).

Therefore, the scheduler enqueue and dequeue operations have to be run from the same thread, which allows the queues and the bitmap operations to be non-thread safe and keeps the scheduler data structures internal to the same core.

Performance Scaling

Scaling up the number of NIC ports simply requires a proportional increase in the number of CPU cores to be used for traffic scheduling.

Enqueue Pipeline

The sequence of steps per packet:

1. *Access* the mbuf to read the data fields required to identify the destination queue for the packet. These fields are: port, subport, traffic class and queue within traffic class, and are typically set by the classification stage.
2. *Access* the queue structure to identify the write location in the queue array. If the queue is full, then the packet is discarded.
3. *Access* the queue array location to store the packet (i.e. write the mbuf pointer).

It should be noted the strong data dependency between these steps, as steps 2 and 3 cannot start before the result from steps 1 and 2 becomes available, which prevents the processor out of order execution engine to provide any significant performance optimizations.

Given the high rate of input packets and the large amount of queues, it is expected that the data structures accessed to enqueue the current packet are not present in the L1 or L2 data cache of the current core, thus the above 3 memory accesses would result (on average) in L1 and L2 data cache misses. A number of 3 L1/L2 cache misses per packet is not acceptable for performance reasons.

The workaround is to prefetch the required data structures in advance. The prefetch operation has an execution latency during which the processor should not attempt to access the data structure currently under prefetch, so the processor should execute other work. The only other work available is to execute different stages of the enqueue sequence of operations on other input packets, thus resulting in a pipelined implementation for the enqueue operation.

Fig. 5.72 illustrates a pipelined implementation for the enqueue operation with 4 pipeline stages and each stage executing 2 different input packets. No input packet can be part of more than one pipeline stage at a given time.

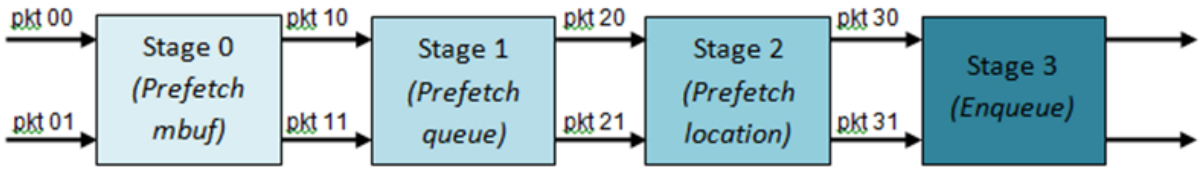


Fig. 5.72: Prefetch Pipeline for the Hierarchical Scheduler Enqueue Operation

The congestion management scheme implemented by the enqueue pipeline described above is very basic: packets are enqueued until a specific queue becomes full, then all the packets destined to the same queue are dropped until packets are consumed (by the dequeue operation). This can be improved by enabling RED/WRED as part of the enqueue pipeline which looks at the queue occupancy and packet priority in order to yield the enqueue/drop decision for a specific packet (as opposed to enqueuing all packets / dropping all packets indiscriminately).

Dequeue State Machine

The sequence of steps to schedule the next packet from the current pipe is:

1. Identify the next active pipe using the bitmap scan operation, *prefetch* pipe.
2. *Read* pipe data structure. Update the credits for the current pipe and its subport. Identify the first active traffic class within the current pipe, select the next queue using WRR, *prefetch* queue pointers for all the 16 queues of the current pipe.
3. *Read* next element from the current WRR queue and *prefetch* its packet descriptor.
4. *Read* the packet length from the packet descriptor (mbuf structure). Based on the packet length and the available credits (of current pipe, pipe traffic class, subport and subport traffic class), take the go/no go scheduling decision for the current packet.

To avoid the cache misses, the above data structures (pipe, queue, queue array, mbufs) are prefetched in advance of being accessed. The strategy of hiding the latency of the prefetch operations is to switch from the current pipe (in grinder A) to another pipe (in grinder B) immediately after a prefetch is issued for the current pipe. This gives enough time to the prefetch operation to complete before the execution switches back to this pipe (in grinder A).

The dequeue pipe state machine exploits the data presence into the processor cache, therefore it tries to send as many packets from the same pipe TC and pipe as possible (up to the available packets and credits) before moving to the next active TC from the same pipe (if any) or to another active pipe.

Timing and Synchronization

The output port is modeled as a conveyor belt of byte slots that need to be filled by the scheduler with data for transmission. For 10 GbE, there are 1.25 billion byte slots that need to be filled by the port scheduler every second. If the scheduler is not fast enough to fill the slots, provided that enough packets and credits exist, then some slots will be left unused and bandwidth will be wasted.

In principle, the hierarchical scheduler dequeue operation should be triggered by NIC TX. Usually, once the occupancy of the NIC TX input queue drops below a predefined threshold, the port scheduler is woken up (interrupt based or polling based, by continuously monitoring the queue occupancy) to push more packets into the queue.

Internal Time Reference

The scheduler needs to keep track of time advancement for the credit logic, which requires credit updates based on time (for example, subport and pipe traffic shaping, traffic class upper limit enforcement, and so on).

Every time the scheduler decides to send a packet out to the NIC TX for transmission, the scheduler will increment its internal time reference accordingly. Therefore, it is convenient to keep the internal time reference in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium. This way, as a packet is scheduled for transmission, the time is incremented with (n + h), where n is the packet length in bytes and h is the number of framing overhead bytes per packet.

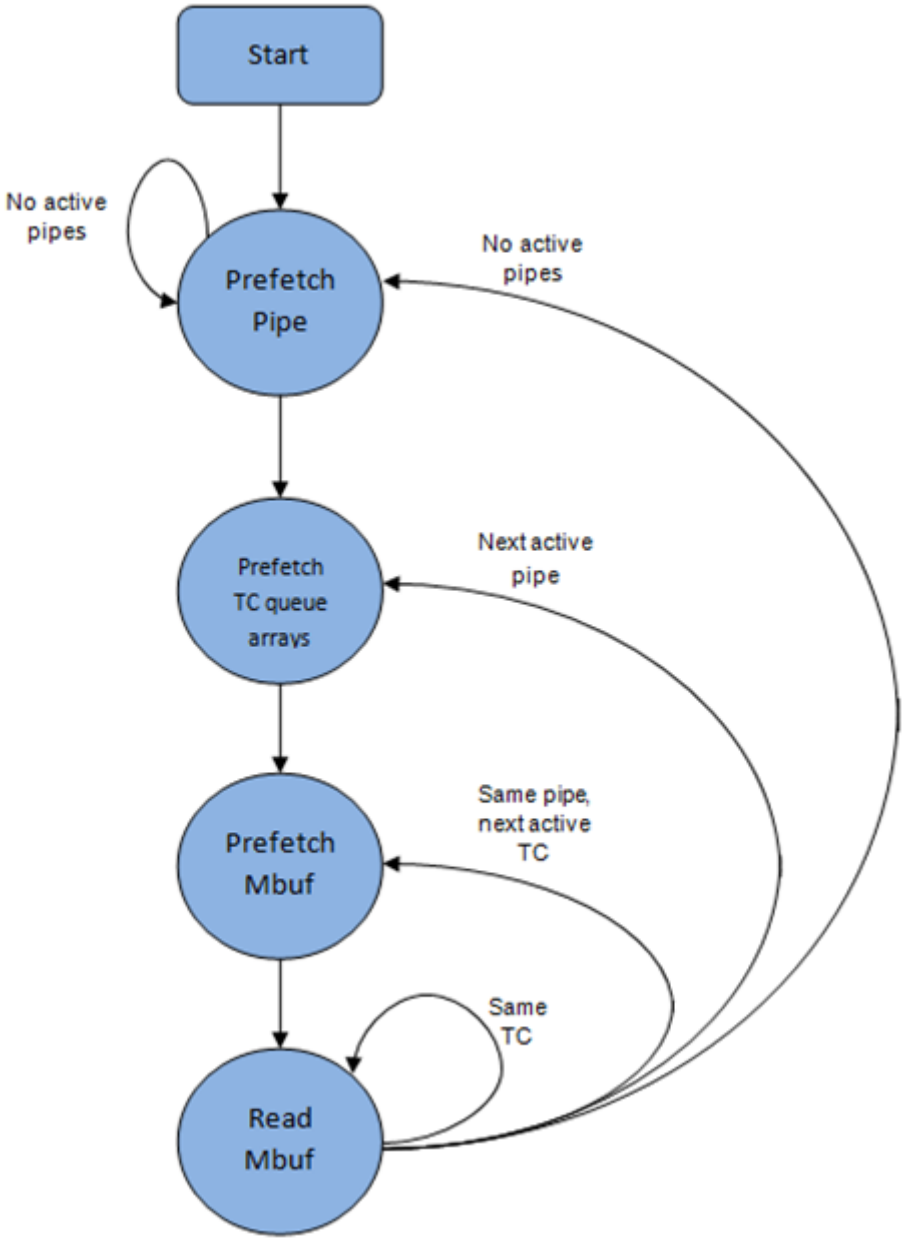


Fig. 5.73: Pipe Prefetch State Machine for the Hierarchical Scheduler Dequeue Operation

Internal Time Reference Re-synchronization

The scheduler needs to align its internal time reference to the pace of the port conveyor belt. The reason is to make sure that the scheduler does not feed the NIC TX with more bytes than the line rate of the physical medium in order to prevent packet drop (by the scheduler, due to the NIC TX input queue being full, or later on, internally by the NIC TX).

The scheduler reads the current time on every dequeue invocation. The CPU time stamp can be obtained by reading either the Time Stamp Counter (TSC) register or the High Precision Event Timer (HPET) register. The current CPU time stamp is converted from number of CPU clocks to number of bytes: *time_bytes = time_cycles / cycles_per_byte, where cycles_per_byte* is the amount of CPU cycles that is equivalent to the transmission time for one byte on the wire (e.g. for a CPU frequency of 2 GHz and a 10GbE port,*cycles_per_byte = 1.6*).

The scheduler maintains an internal time reference of the NIC time. Whenever a packet is scheduled, the NIC time is incremented with the packet length (including framing overhead). On every dequeue invocation, the scheduler checks its internal reference of the NIC time against the current time:

1. If NIC time is in the future (NIC time >= current time), no adjustment of NIC time is needed. This means that scheduler is able to schedule NIC packets before the NIC actually needs those packets, so the NIC TX is well supplied with packets;
2. If NIC time is in the past (NIC time < current time), then NIC time should be adjusted by setting it to the current time. This means that the scheduler is not able to keep up with the speed of the NIC byte conveyor belt, so NIC bandwidth is wasted due to poor packet supply to the NIC TX.

Scheduler Accuracy and Granularity

The scheduler round trip delay (SRTD) is the time (number of CPU cycles) between two consecutive examinations of the same pipe by the scheduler.

To keep up with the output port (that is, avoid bandwidth loss), the scheduler should be able to schedule n packets faster than the same n packets are transmitted by NIC TX.

The scheduler needs to keep up with the rate of each individual pipe, as configured for the pipe token bucket, assuming that no port oversubscription is taking place. This means that the size of the pipe token bucket should be set high enough to prevent it from overflowing due to big SRTD, as this would result in credit loss (and therefore bandwidth loss) for the pipe.

Credit Logic

Scheduling Decision

The scheduling decision to send next packet from (subport S, pipe P, traffic class TC, queue Q) is favorable (packet is sent) when all the conditions below are met:

- Pipe P of subport S is currently selected by one of the port grinders;
- Traffic class TC is the highest priority active traffic class of pipe P;
- Queue Q is the next queue selected by WRR within traffic class TC of pipe P;
- Subport S has enough credits to send the packet;

- Subport S has enough credits for traffic class TC to send the packet;
- Pipe P has enough credits to send the packet;
- Pipe P has enough credits for traffic class TC to send the packet.

If all the above conditions are met, then the packet is selected for transmission and the necessary credits are subtracted from subport S, subport S traffic class TC, pipe P, pipe P traffic class TC.

Framing Overhead

As the greatest common divisor for all packet lengths is one byte, the unit of credit is selected as one byte. The number of credits required for the transmission of a packet of n bytes is equal to (n+h), where h is equal to the number of framing overhead bytes per packet.

Table 5.100: Ethernet Frame Overhead Fields

#	Packet field	Length (bytes)	Comments
1	Preamble	7	
2	Start of Frame Delimiter (SFD)	1	
3	Frame Check Sequence (FCS)	4	Considered overhead only if not included in the mbuf packet length field.
4	Inter Frame Gap (IFG)	12	
5	Total	24	

Traffic Shaping

The traffic shaping for subport and pipe is implemented using a token bucket per subport/per pipe. Each token bucket is implemented using one saturated counter that keeps track of the number of available credits.

The token bucket generic parameters and operations are presented in Table 5.101 and Table 5.102.

Table 5.101: Token Bucket Generic Parameters

#	Token Bucket Parameter	Unit	Description
1	bucket_rate	Credits per second	Rate of adding credits to the bucket.
2	bucket_size	Credits	Max number of credits that can be stored in the bucket.

Table 5.102: Token Bucket Generic Operations

#	Token Bucket Operation	Description
1	Initialization	Bucket set to a predefined value, e.g. zero or half of the bucket size.
2	Credit update	Credits are added to the bucket on top of existing ones, either periodically or on demand, based on the bucket_rate. Credits cannot exceed the upper limit defined by the bucket_size, so any credits to be added to the bucket while the bucket is full are dropped.
3	Credit consumption	As result of packet scheduling, the necessary number of credits is removed from the bucket. The packet can only be sent if enough credits are in the bucket to send the full packet (packet bytes and framing overhead for the packet).

To implement the token bucket generic operations described above, the current design uses the persistent data structure presented in Table 5.103, while the implementation of the token bucket operations is described in Table 5.104.

Table 5.103: Token Bucket Persistent Data Structure

#	Token bucket field	Unit	Description
1	tb_time	Bytes	Time of the last credit update. Measured in bytes instead of seconds or CPU cycles for ease of credit consumption operation (as the current time is also maintained in bytes). See Section 26.2.4.5.1 “Internal Time Reference” for an explanation of why the time is maintained in byte units.
2	tb_period	Bytes	Time period that should elapse since the last credit update in order for the bucket to be awarded tb_credits_per_period worth or credits.
3	tb_credits_per_period	Bytes	Credit allowance per tb_period.
4	tb_size	Bytes	Bucket size, i.e. upper limit for the tb_credits.
5	tb_credits	Bytes	Number of credits currently in the bucket.

The bucket rate (in bytes per second) can be computed with the following formula:

*bucket_rate = (tb_credits_per_period / tb_period) * r*

where, r = port line rate (in bytes per second).

Table 5.104: Token Bucket Operations

#	Token bucket operation	Description
1	Initialization	*tb_credits = 0; or tb_credits = tb_size / 2;*
2	Credit update	Credit update options: • Every time a packet is sent for a port, update the credits of all the the subports and pipes of that port. Not feasible. • Every time a packet is sent, update the credits for the pipe and subport. Very accurate, but not needed (a lot of calculations). • Every time a pipe is selected (that is, picked by one of the grinders), update the credits for the pipe and its subport. The current implementation is using option 3. According to Section *Dequeue State Machine*, the pipe and subport credits are updated every time a pipe is selected by the dequeue process before the pipe and subport credits are actually used. The implementation uses a tradeoff between accuracy and speed by updating the bucket credits only when at least a full *tb_period* has elapsed since the last update. • Full accuracy can be achieved by selecting the value for *tb_period* for which *tb_credits_per_period = 1*. • When full accuracy is not required, better performance is achieved by setting *tb_credits* to a larger value. Update operations: • n_periods = (time - tb_time) / tb_period; • tb_credits += n_periods * tb_credits_per_period; • tb_credits = min(tb_credits, tb_size); • tb_time += n_periods * tb_period;

Traffic Classes

Implementation of Strict Priority Scheduling

Strict priority scheduling of traffic classes within the same pipe is implemented by the pipe dequeue state machine, which selects the queues in ascending order. Therefore, queue 0 (associated with TC 0, highest priority TC) is handled before queue 1 (TC 1, lower priority than TC 0), which is handled before queue 2 (TC 2, lower priority than TC 1) and it conitnues until queues of all TCs except the lowest priority TC are handled. At last, queues 12..15 (best effort TC, lowest priority TC) are handled.

Upper Limit Enforcement

The traffic classes at the pipe and subport levels are not traffic shaped, so there is no token bucket maintained in this context. The upper limit for the traffic classes at the subport and pipe levels is enforced by periodically refilling the subport / pipe traffic class credit counter, out of which credits are consumed every time a packet is scheduled for that subport / pipe, as described in Table 5.105 and Table 5.106.

Table 5.105: Subport/Pipe Traffic Class Upper Limit Enforcement Persistent Data Structure

#	Subport or pipe field	Unit	Description
1	tc_time	Bytes	Time of the next update (upper limit refill) for the TCs of the current subport / pipe. See Section *Internal Time Reference* for the explanation of why the time is maintained in byte units.
2	tc_period	Bytes	Time between two consecutive updates for the all TCs of the current subport / pipe. This is expected to be many times bigger than the typical value of the token bucket tb_period.
3	tc_credits_per_period	Bytes	Upper limit for the number of credits allowed to be consumed by the current TC during each enforcement period tc_period.
4	tc_credits	Bytes	Current upper limit for the number of credits that can be consumed by the current traffic class for the remainder of the current enforcement period.

Table 5.106: Subport/Pipe Traffic Class Upper Limit Enforcement Operations

#	Traffic Class Operation	Description
1	Initialization	tc_credits = tc_credits_per_period; tc_time = tc_period;

2	Credit update	Update operations:
		if (time >= tc_time) {
		tc_credits = tc_credits_per_period;
		tc_time = time + tc_period;
		}

3	Credit consumption (on packet scheduling)	As result of packet scheduling, the TC limit is decreased with the necessary number of credits. The packet can only be sent if enough credits are currently available in the TC limit to send the full packet (packet bytes and framing overhead for the packet).
		Scheduling operations:
		pkt_credits = pk_len + frame_overhead;
		if (tc_credits >= pkt_credits) {tc_credits -= pkt_credits;}

Weighted Round Robin (WRR)

The evolution of the WRR design solution for the lowest priority traffic class (best effort TC) from simple to complex is shown in Table 5.107.

Table 5.107: Weighted Round Robin (WRR)

#	All Queues Active?	Equal Weights for All Queues?	All Packets Equal?	Strategy
1	Yes	Yes	Yes	**Byte level round robin** *Next queue* queue #i, i = *(i + 1) % n*
2	Yes	Yes	No	**Packet level round robin** Consuming one byte from queue #i requires consuming exactly one token for queue #i. T(i) = Accumulated number of tokens previously consumed from queue #i. Every time a packet is consumed from queue #i, T(i) is updated as: T(i) += *pkt_len*. *Next queue* : queue with the smallest T.
3	Yes	No	No	**Packet level weighted round robin** This case can be reduced to the previous case by introducing a cost per byte that is different for each queue. Queues with lower weights have a higher cost per byte. This way, it is still meaningful to compare the consumption amongst different queues in order to select the next queue. w(i) = Weight of queue #i t(i) = Tokens per byte for queue #i, defined as the inverse weight of queue #i. For example, if w[0..3] = [1:2:4:8], then t[0..3] = [8:4:2:1]; if w[0..3] = [1:4:15:20], then t[0..3] = [60:15:4:3]. Consuming one byte from queue #i requires consuming t(i) tokens for queue #i. T(i) = Accumulated number of tokens previously consumed from queue #i. Every time a packet is consumed from queue #i, T(i) is updated as: *T(i) += pkt_len * t(i). Next queue* : queue with the smallest T.
4	No	No	No	**Packet level weighted round robin with variable queue status** Reduce this case to the previous case by setting the consumption of inactive queues to a high number, so that the inactive queues will never be selected by the smallest T logic. To prevent T from overflowing as result of successive accumulations, T(i) is truncated after each packet consumption for all queues. For example, T[0..3] = [1000, 1100, 1200, 1300] is truncated to T[0..3] = [0, 100, 200, 300] by subtracting the min T from T(i), i = 0..n. This requires having at least one active queue in the set of input queues, which is guaranteed by the dequeue state machine never selecting an inactive traffic class. *mask(i) = Saturation mask for queue #i, defined as:* mask(i) = (queue #i is active)? 0 : 0xFFFFFFFF; w(i) = Weight of queue #i t(i) = Tokens per byte for queue #i, defined as the inverse weight of queue #i. T(i) = Accumulated numbers of tokens previously consumed from queue #i. *Next queue* : queue with smallest T. Before packet consumption from queue #i: *T(i) |= mask(i)* After packet consumption from queue #i: T(j) -= T(i), j != i T(i) = pkt_len * t(i) Note: T(j) uses the T(i) value before T(i) is updated.

Subport Traffic Class Oversubscription

Problem Statement

Oversubscription for subport traffic class X is a configuration-time event that occurs when more bandwidth is allocated for traffic class X at the level of subport member pipes than allocated for the same traffic class at the parent subport level.

The existence of the oversubscription for a specific subport and traffic class is solely the result of pipe and subport-level configuration as opposed to being created due to dynamic evolution of the traffic load at run-time (as congestion is).

When the overall demand for traffic class X for the current subport is low, the existence of the oversubscription condition does not represent a problem, as demand for traffic class X is completely satisfied for all member pipes. However, this can no longer be achieved when the aggregated demand for traffic class X for all subport member pipes exceeds the limit configured at the subport level.

Solution Space

summarizes some of the possible approaches for handling this problem, with the third approach selected for implementation.

Table 5.108: Subport Traffic Class Oversubscription

No.	Approach	Description
1	Don't care	First come, first served. This approach is not fair amongst subport member pipes, as pipes that are served first will use up as much bandwidth for TC X as they need, while pipes that are served later will receive poor service due to bandwidth for TC X at the subport level being scarce.
2	Scale down all pipes	All pipes within the subport have their bandwidth limit for TC X scaled down by the same factor. This approach is not fair among subport member pipes, as the low end pipes (that is, pipes configured with low bandwidth) can potentially experience severe service degradation that might render their service unusable (if available bandwidth for these pipes drops below the minimum requirements for a workable service), while the service degradation for high end pipes might not be noticeable at all.
3	Cap the high demand pipes	Each subport member pipe receives an equal share of the bandwidth available at run-time for TC X at the subport level. Any bandwidth left unused by the low-demand pipes is redistributed in equal portions to the high-demand pipes. This way, the high-demand pipes are truncated while the low-demand pipes are not impacted.

Typically, the subport TC oversubscription feature is enabled only for the lowest priority traffic class, which is typically used for best effort traffic, with the management plane preventing this condition from occurring for the other (higher priority) traffic classes.

To ease implementation, it is also assumed that the upper limit for subport best effort TC is set to 100% of the subport rate, and that the upper limit for pipe best effort TC is set to 100% of pipe rate for all subport member pipes.

Implementation Overview

The algorithm computes a watermark, which is periodically updated based on the current demand experienced by the subport member pipes, whose purpose is to limit the amount of traffic that each pipe is allowed to send for best effort TC. The watermark is computed at the subport level at the beginning of each traffic class upper limit enforcement period and the same value is used by all the subport member pipes throughout the current enforcement period. illustrates how the watermark computed as subport level at the beginning of each period is propagated to all subport member pipes.

At the beginning of the current enforcement period (which coincides with the end of the previous enforcement period), the value of the watermark is adjusted based on the amount of bandwidth allocated to best effort TC at the beginning of the previous period that was not left unused by the subport member pipes at the end of the previous period.

If there was subport best effort TC bandwidth left unused, the value of the watermark for the current period is increased to encourage the subport member pipes to consume more bandwidth. Otherwise, the value of the watermark is decreased to enforce equality of bandwidth consumption among subport member pipes for best effort TC.

The increase or decrease in the watermark value is done in small increments, so several enforcement periods might be required to reach the equilibrium state. This state can change at any moment due to variations in the demand experienced by the subport member pipes for best effort TC, for example, as a result of demand increase (when the watermark needs to be lowered) or demand decrease (when the watermark needs to be increased).

When demand is low, the watermark is set high to prevent it from impeding the subport member pipes from consuming more bandwidth. The highest value for the watermark is picked as the highest rate configured for a subport member pipe. Table 5.109 and Table 5.110 illustrates the watermark operation.

Table 5.109: Watermark Propagation from Subport Level to Member Pipes at the Beginning of Each Traffic Class Upper Limit Enforcement Period

No.	Subport Traffic Class Operation	Description
1	Initialization	**Subport level**: subport_period_id= 0 **Pipe level**: pipe_period_id = 0

2	Credit update	**Subport Level**:
		if (time>=subport_tc_time)
		{
		subport_wm = water_mark_update();
		subport_tc_time = time + subport_tc_period;
		subport_period_id++;
		}
		Pipelevel:
		if(pipe_period_id != subport_period_id)
		{
		pipe_ov_credits = subport_wm * pipe_weight;
		pipe_period_id = subport_period_id;
		}

3	Credit consumption (on packet scheduling)	**Pipe level:**
		pkt_credits = pk_len + frame_overhead;
		if(pipe_ov_credits >= pkt_credits{
		pipe_ov_credits -= pkt_credits;
		}

Table 5.110: Watermark Calculation

No.	Subport Traffic Class Operation	Description
1	Initialization	**Subport level:** wm = WM_MAX

2	Credit update	**Subport level (water_mark_update):**
		tc0_cons = subport_tc0_credits_per_period - subport_tc0_credits;
		tc1_cons = subport_tc1_credits_per_period - subport_tc1_credits;
		tc2_cons = subport_tc2_credits_per_period - subport_tc2_credits;
		tc3_cons = subport_tc3_credits_per_period - subport_tc3_credits;
		tc4_cons = subport_tc4_credits_per_period - subport_tc4_credits;
		tc5_cons = subport_tc5_credits_per_period - subport_tc5_credits;
		tc6_cons = subport_tc6_credits_per_period - subport_tc6_credits;
		tc7_cons = subport_tc7_credits_per_period - subport_tc7_credits;
		tc8_cons = subport_tc8_credits_per_period - subport_tc8_credits;
		tc9_cons = subport_tc9_credits_per_period - subport_tc9_credits;
		tc10_cons = subport_tc10_credits_per_period - subport_tc10_credits;
		tc11_cons = subport_tc11_credits_per_period - subport_tc11_credits;
		tc_be_cons_max = subport_tc_be_credits_per_period - (tc0_cons + tc1_cons + tc2_cons + tc3_cons + tc4_cons + tc5_cons + tc6_cons + tc7_cons + tc8_cons + tc9_cons + tc10_cons + tc11_cons);
		if(tc_be_consumption > (tc_be_consumption_max - MTU)){
		wm -= wm >> 7;
		if(wm < WM_MIN) wm =

Worst Case Scenarios for Performance

Lots of Active Queues with Not Enough Credits

The more queues the scheduler has to examine for packets and credits in order to select one packet, the lower the performance of the scheduler is.

The scheduler maintains the bitmap of active queues, which skips the non-active queues, but in order to detect whether a specific pipe has enough credits, the pipe has to be drilled down using the pipe dequeue state machine, which consumes cycles regardless of the scheduling result (no packets are produced or at least one packet is produced).

This scenario stresses the importance of the policer for the scheduler performance: if the pipe does not have enough credits, its packets should be dropped as soon as possible (before they reach the hierarchical scheduler), thus rendering the pipe queues as not active, which allows the dequeue side to skip that pipe with no cycles being spent on investigating the pipe credits that would result in a "not enough credits" status.

Single Queue with 100% Line Rate

The port scheduler performance is optimized for a large number of queues. If the number of queues is small, then the performance of the port scheduler for the same level of active traffic is expected to be worse than the performance of a small set of message passing queues.

5.43.3 Dropper

The purpose of the DPDK dropper is to drop packets arriving at a packet scheduler to avoid congestion. The dropper supports the Random Early Detection (RED), Weighted Random Early Detection (WRED) and tail drop algorithms. Fig. 5.74 illustrates how the dropper integrates with the scheduler. The DPDK currently does not support congestion management so the dropper provides the only method for congestion avoidance.

The dropper uses the Random Early Detection (RED) congestion avoidance algorithm as documented in the reference publication. The purpose of the RED algorithm is to monitor a packet queue, determine the current congestion level in the queue and decide whether an arriving packet should be enqueued or dropped. The RED algorithm uses an Exponential Weighted Moving Average (EWMA) filter to compute average queue size which gives an indication of the current congestion level in the queue.

For each enqueue operation, the RED algorithm compares the average queue size to minimum and maximum thresholds. Depending on whether the average queue size is below, above or in between these thresholds, the RED algorithm calculates the probability that an arriving packet should be dropped and makes a random decision based on this probability.

The dropper also supports Weighted Random Early Detection (WRED) by allowing the scheduler to select different RED configurations for the same packet queue at run-time. In the case of severe congestion, the dropper resorts to tail drop. This occurs when a packet queue has reached maximum capacity and cannot store any more packets. In this situation, all arriving packets are dropped.

The flow through the dropper is illustrated in Fig. 5.75. The RED/WRED algorithm is exercised first and tail drop second.

The use cases supported by the dropper are:

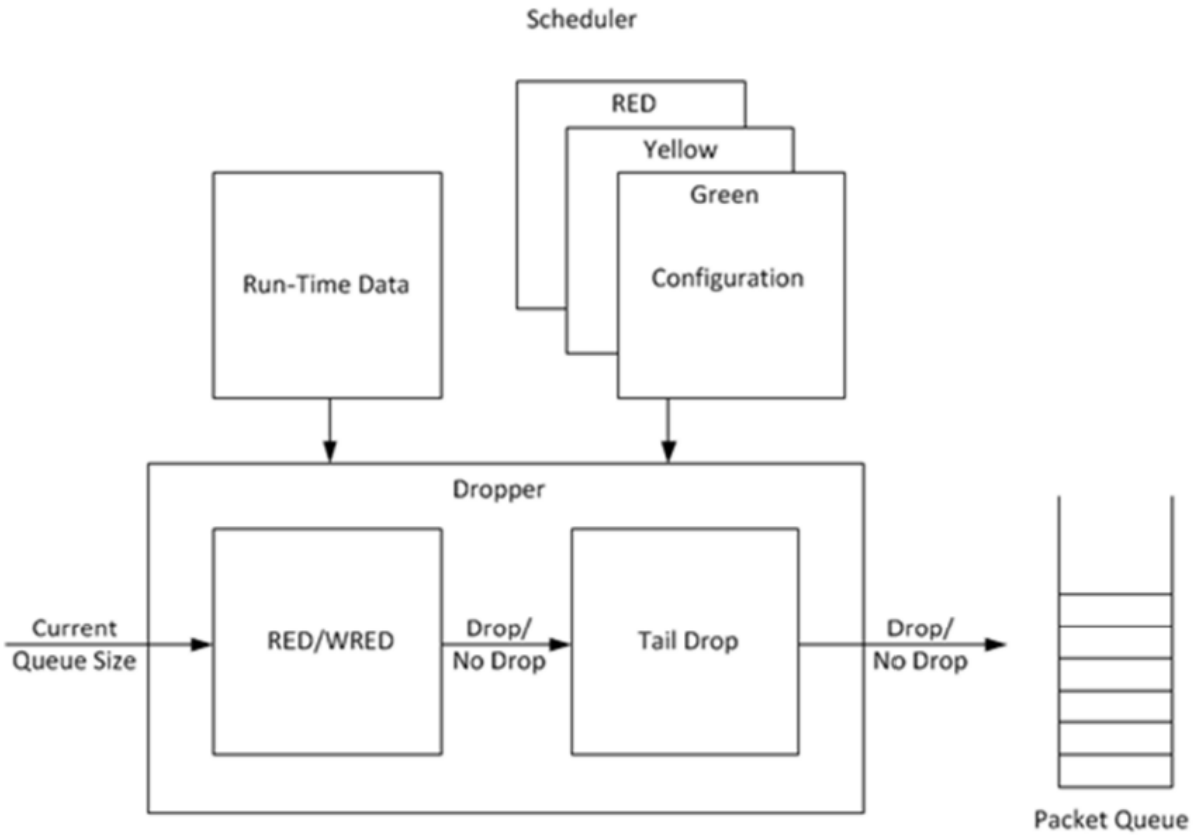


Fig. 5.74: High-level Block Diagram of the DPDK Dropper

- – Initialize configuration data
- – Initialize run-time data
- – Enqueue (make a decision to enqueue or drop an arriving packet)
- – Mark empty (record the time at which a packet queue becomes empty)

The configuration use case is explained in *Section2.23.3.1*, the enqueue operation is explained in *Section 2.23.3.2* and the mark empty operation is explained in *Section 2.23.3.3*.

Configuration

A RED configuration contains the parameters given in Table 5.111.

Table 5.111: RED Configuration Parameters

Parameter	Minimum	Maximum	Typical
Minimum Threshold	0	1022	1/4 x queue size
Maximum Threshold	1	1023	1/2 x queue size
Inverse Mark Probability	1	255	10
EWMA Filter Weight	1	12	9

The meaning of these parameters is explained in more detail in the following sections. The format of these parameters as specified to the dropper module API corresponds to the format used by Cisco* in their RED implementation. The minimum and maximum threshold parameters are specified to the dropper

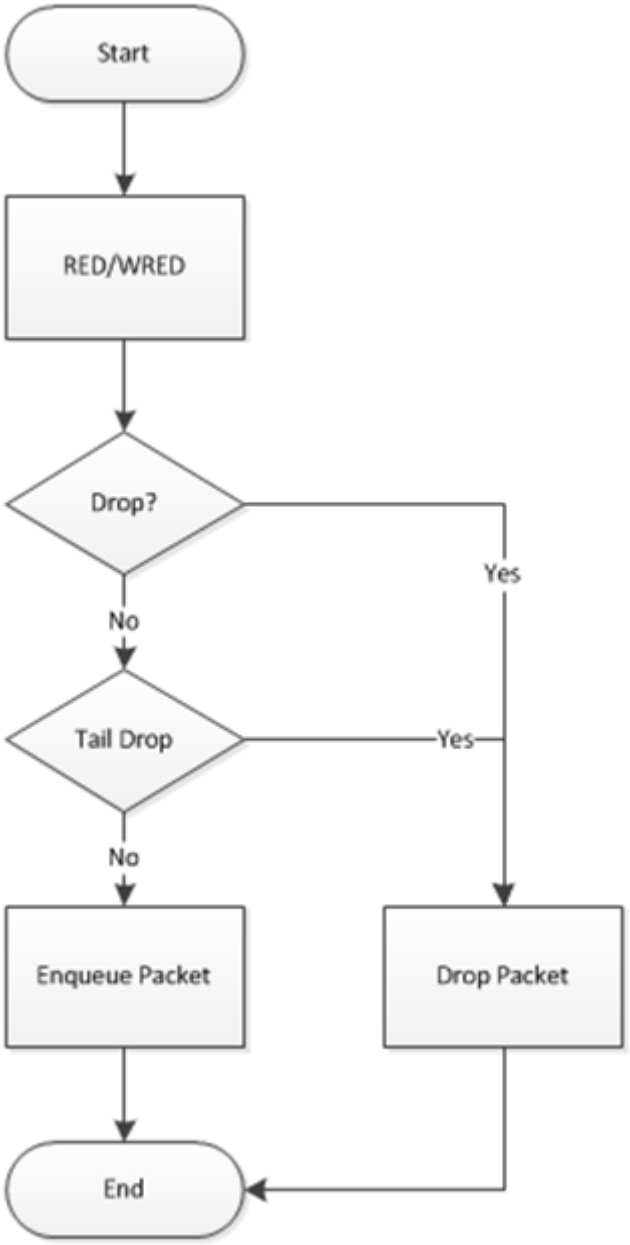


Fig. 5.75: Flow Through the Dropper

module in terms of number of packets. The mark probability parameter is specified as an inverse value, for example, an inverse mark probability parameter value of 10 corresponds to a mark probability of 1/10 (that is, 1 in 10 packets will be dropped). The EWMA filter weight parameter is specified as an inverse log value, for example, a filter weight parameter value of 9 corresponds to a filter weight of 1/29.

Enqueue Operation

In the example shown in Fig. 5.76, q (actual queue size) is the input value, avg (average queue size) and count (number of packets since the last drop) are run-time values, decision is the output value and the remaining values are configuration parameters.

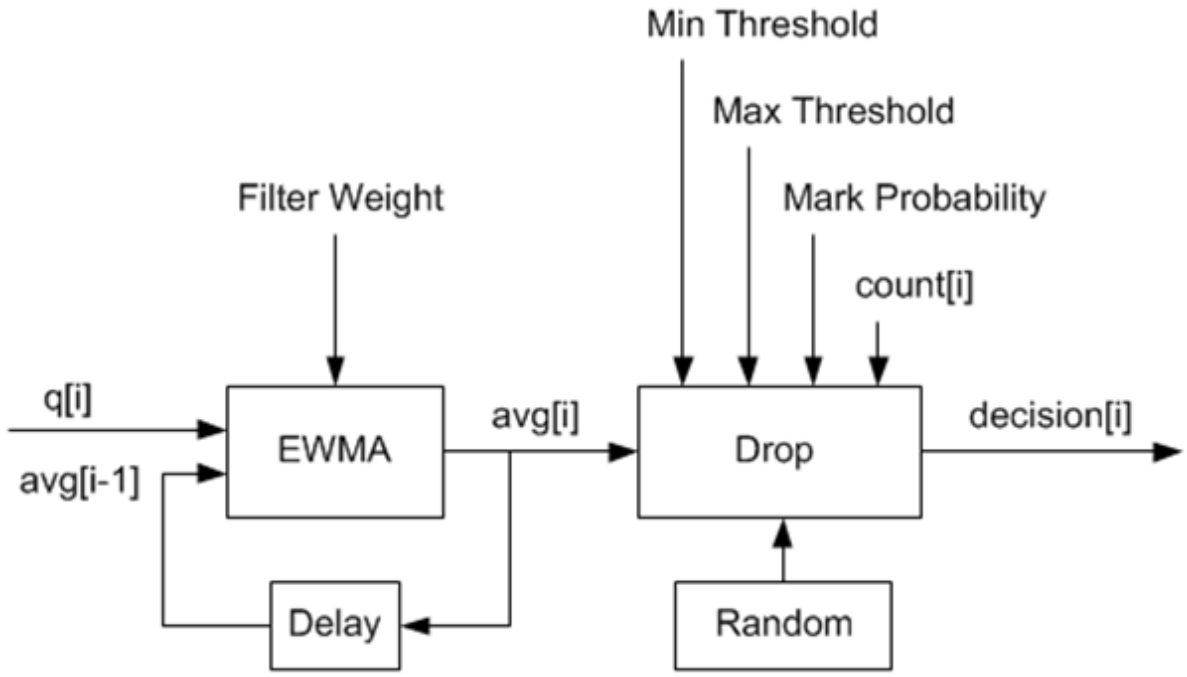


Fig. 5.76: Example Data Flow Through Dropper

EWMA Filter Microblock

The purpose of the EWMA Filter microblock is to filter queue size values to smooth out transient changes that result from "bursty" traffic. The output value is the average queue size which gives a more stable view of the current congestion level in the queue.

The EWMA filter has one configuration parameter, filter weight, which determines how quickly or slowly the average queue size output responds to changes in the actual queue size input. Higher values of filter weight mean that the average queue size responds more quickly to changes in actual queue size.

Average Queue Size Calculation when the Queue is not Empty

The definition of the EWMA filter is given in the following equation.

$$avg[i] = (1 - w_q) \times avg[i-1] + w_q \times q[i]$$

Where:

- *avg* = average queue size
- *wq* = filter weight
- *q* = actual queue size

Note:

The filter weight, wq = 1/2^n, where n is the filter weight parameter value passed to the dropper module
on configuration (see *Section2.23.3.1*).

Average Queue Size Calculation when the Queue is Empty

The EWMA filter does not read time stamps and instead assumes that enqueue operations will happen quite regularly. Special handling is required when the queue becomes empty as the queue could be empty for a short time or a long time. When the queue becomes empty, average queue size should decay gradually to zero instead of dropping suddenly to zero or remaining stagnant at the last computed value. When a packet is enqueued on an empty queue, the average queue size is computed using the following formula:

$$avg[i] = \text{avg}[i-1] \times (1 - \text{w}_\text{q})^m$$

Where:

- *m* = the number of enqueue operations that could have occurred on this queue while the queue was empty

In the dropper module, *m* is defined as:

$$m = \left(\frac{time - qtime}{s}\right)$$

Where:

- *time* = current time
- *qtime* = time the queue became empty
- *s* = typical time between successive enqueue operations on this queue

The time reference is in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium (see Section *Internal Time Reference*). The parameter s is defined in the dropper module as a constant with the value: s=2^22. This corresponds to the time required by every leaf node in a hierarchy with 64K leaf nodes to transmit one 64-byte packet onto the wire and represents the worst case scenario. For much smaller scheduler hierarchies, it may be necessary to reduce the parameter s, which is defined in the red header source file (rte_red.h) as:

```
#define RTE_RED_S
```

Since the time reference is in bytes, the port speed is implied in the expression: *time-qtime*. The dropper does not have to be configured with the actual port speed. It adjusts automatically to low speed and high speed links.

Implementation

A numerical method is used to compute the factor (1-wq)^m that appears in Equation 2.

This method is based on the following identity:

$$a \equiv 2^{(b \times \log_2(a))}$$

This allows us to express the following:

$$(1 - w_q)^m = 2^{(m \times \log_2(1 - w_q))}$$

In the dropper module, a look-up table is used to compute log2(1-wq) for each value of wq supported by the dropper module. The factor (1-wq)^m can then be obtained by multiplying the table value by *m* and applying shift operations. To avoid overflow in the multiplication, the value, *m*, and the look-up table values are limited to 16 bits. The total size of the look-up table is 56 bytes. Once the factor (1-wq)^m is obtained using this method, the average queue size can be calculated from Equation 2.

Alternative Approaches

Other methods for calculating the factor (1-wq)^m in the expression for computing average queue size when the queue is empty (Equation 2) were considered. These approaches include:

- Floating-point evaluation
- Fixed-point evaluation using a small look-up table (512B) and up to 16 multiplications (this is the approach used in the FreeBSD* ALTQ RED implementation)
- Fixed-point evaluation using a small look-up table (512B) and 16 SSE multiplications (SSE optimized version of the approach used in the FreeBSD* ALTQ RED implementation)
- Large look-up table (76 KB)

The method that was finally selected (described above in Section 26.3.2.2.1) out performs all of these approaches in terms of run-time performance and memory requirements and also achieves accuracy comparable to floating-point evaluation. Table 5.112 lists the performance of each of these alternative approaches relative to the method that is used in the dropper. As can be seen, the floating-point implementation achieved the worst performance.

Table 5.112: Relative Performance of Alternative Approaches

Method	Relative Performance
Current dropper method (see *Section 23.3.2.1.3*)	100%
Fixed-point method with small (512B) look-up table	148%
SSE method with small (512B) look-up table	114%
Large (76KB) look-up table	118%
Floating-point	595%
Note: In this case, since performance is expressed as time spent executing the operation in a specific condition, any relative performance value above 100% runs slower than the reference method.	

Drop Decision Block

The Drop Decision block:

- Compares the average queue size with the minimum and maximum thresholds
- Calculates a packet drop probability
- Makes a random decision to enqueue or drop an arriving packet

The calculation of the drop probability occurs in two stages. An initial drop probability is calculated based on the average queue size, the minimum and maximum thresholds and the mark probability. An actual drop probability is then computed from the initial drop probability. The actual drop probability takes the count run-time value into consideration so that the actual drop probability increases as more packets arrive to the packet queue since the last packet was dropped.

Initial Packet Drop Probability

The initial drop probability is calculated using the following equation.

$$p_b = \begin{cases} 0, & avg < min_{th} \\ max_p\left(\frac{avg - min_{th}}{max_{th} - min_{th}}\right), & min_{th} \leq avg < max_{th} \\ 1, & avg \geq max_{th} \end{cases}$$

Where:

- *maxp* = mark probability
- *avg* = average queue size
- *minth* = minimum threshold
- *maxth* = maximum threshold

The calculation of the packet drop probability using Equation 3 is illustrated in Fig. 5.77. If the average queue size is below the minimum threshold, an arriving packet is enqueued. If the average queue size is at or above the maximum threshold, an arriving packet is dropped. If the average queue size is between the minimum and maximum thresholds, a drop probability is calculated to determine if the packet should be enqueued or dropped.

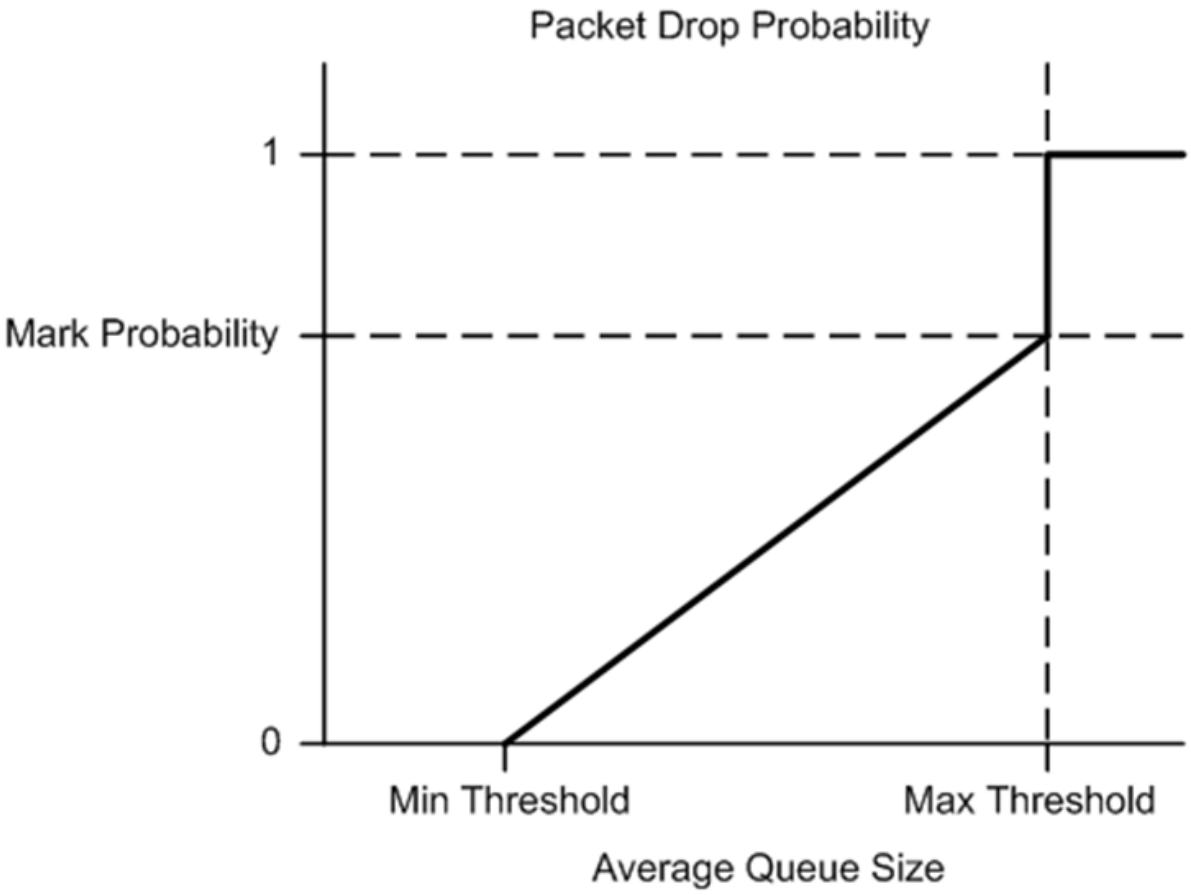


Fig. 5.77: Packet Drop Probability for a Given RED Configuration

Actual Drop Probability

If the average queue size is between the minimum and maximum thresholds, then the actual drop probability is calculated from the following equation.

$$p_a = \frac{p_b}{(2 - count \times p_b)}$$

Where:

- *Pb* = initial drop probability (from Equation 3)
- *count* = number of packets that have arrived since the last drop

The constant 2, in Equation 4 is the only deviation from the drop probability formulae given in the reference document where a value of 1 is used instead. It should be noted that the value pa computed from can be negative or greater than 1. If this is the case, then a value of 1 should be used instead.

The initial and actual drop probabilities are shown in Fig. 5.78. The actual drop probability is shown for the case where the formula given in the reference document1 is used (blue curve) and also for the case where the formula implemented in the dropper module, is used (red curve). The formula in the reference document results in a significantly higher drop rate compared to the mark probability configuration parameter specified by the user. The choice to deviate from the reference document is simply a design decision and one that has been taken by other RED implementations, for example, FreeBSD* ALTQ RED.

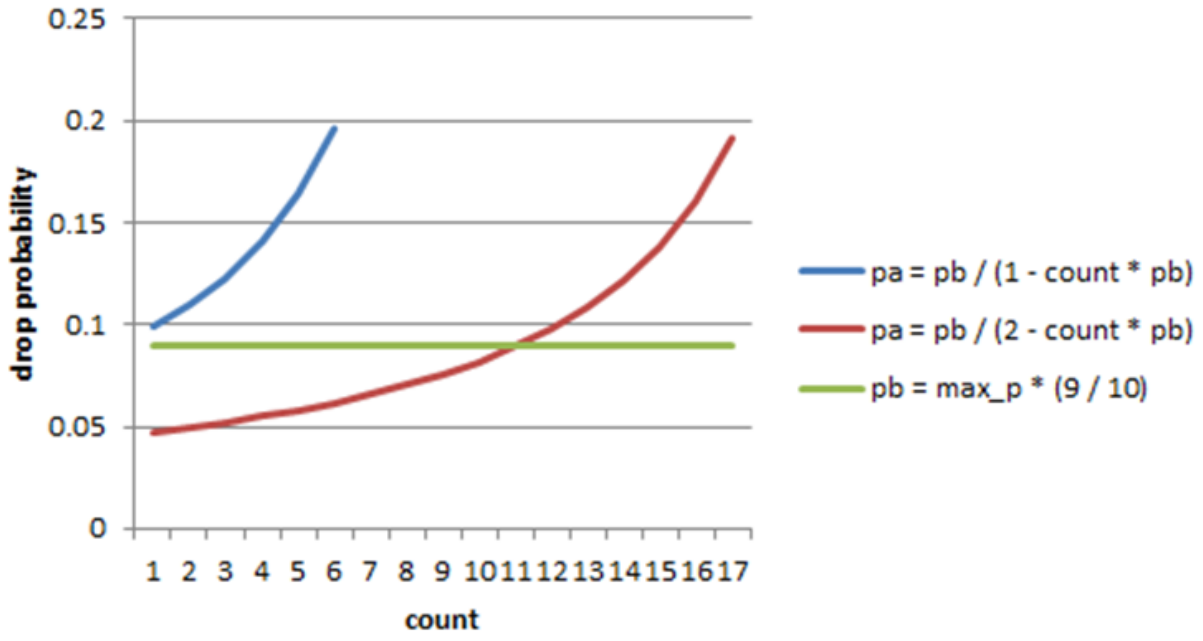


Fig. 5.78: Initial Drop Probability (pb), Actual Drop probability (pa) Computed Using a Factor 1 (Blue Curve) and a Factor 2 (Red Curve)

Queue Empty Operation

The time at which a packet queue becomes empty must be recorded and saved with the RED run-time data so that the EWMA filter block can calculate the average queue size on the next enqueue operation. It is the responsibility of the calling application to inform the dropper module through the API that a queue has become empty.

Source Files Location

The source files for the DPDK dropper are located at:

- DPDK/lib/librte_sched/rte_red.h
- DPDK/lib/librte_sched/rte_red.c

Integration with the DPDK QoS Scheduler

RED functionality in the DPDK QoS scheduler is disabled by default. To enable it, use the DPDK configuration parameter:

```
CONFIG_RTE_SCHED_RED=y
```

This parameter must be set to y. The parameter is found in the build configuration files in the DPDK/config directory, for example, DPDK/config/common_linux. RED configuration parameters are specified in the rte_red_params structure within the rte_sched_port_params structure that is passed to the scheduler on initialization. RED parameters are specified separately for four traffic classes and three packet colors (green, yellow and red) allowing the scheduler to implement Weighted Random Early Detection (WRED).

Integration with the DPDK QoS Scheduler Sample Application

The DPDK QoS Scheduler Application reads a configuration file on start-up. The configuration file includes a section containing RED parameters. The format of these parameters is described in *Section2.23.3.1*. A sample RED configuration is shown below. In this example, the queue size is 64 packets.

Note: For correct operation, the same EWMA filter weight parameter (wred weight) should be used for each packet color (green, yellow, red) in the same traffic class (tc).

```
 ; RED params per traffic class and color (Green / Yellow / Red)

[red]
tc 0 wred min = 28 22 16
tc 0 wred max = 32 32 32
tc 0 wred inv prob = 10 10 10
tc 0 wred weight = 9 9 9

tc 1 wred min = 28 22 16
tc 1 wred max = 32 32 32
tc 1 wred inv prob = 10 10 10
tc 1 wred weight = 9 9 9
```

(continues on next page)

(continued from previous page)

```
tc 2 wred min = 28 22 16
tc 2 wred max = 32 32 32
tc 2 wred inv prob = 10 10 10
tc 2 wred weight = 9 9 9

tc 3 wred min = 28 22 16
tc 3 wred max = 32 32 32
tc 3 wred inv prob = 10 10 10
tc 3 wred weight = 9 9 9

tc 4 wred min = 28 22 16
tc 4 wred max = 32 32 32
tc 4 wred inv prob = 10 10 10
tc 4 wred weight = 9 9 9

tc 5 wred min = 28 22 16
tc 5 wred max = 32 32 32
tc 5 wred inv prob = 10 10 10
tc 5 wred weight = 9 9 9

tc 6 wred min = 28 22 16
tc 6 wred max = 32 32 32
tc 6 wred inv prob = 10 10 10
tc 6 wred weight = 9 9 9

tc 7 wred min = 28 22 16
tc 7 wred max = 32 32 32
tc 7 wred inv prob = 10 10 10
tc 7 wred weight = 9 9 9

tc 8 wred min = 28 22 16
tc 8 wred max = 32 32 32
tc 8 wred inv prob = 10 10 10
tc 8 wred weight = 9 9 9

tc 9 wred min = 28 22 16
tc 9 wred max = 32 32 32
tc 9 wred inv prob = 10 10 10
tc 9 wred weight = 9 9 9


tc 10 wred min = 28 22 16
tc 10 wred max = 32 32 32
tc 10 wred inv prob = 10 10 10
tc 10 wred weight = 9 9 9

tc 11 wred min = 28 22 16
tc 11 wred max = 32 32 32
tc 11 wred inv prob = 10 10 10
tc 11 wred weight = 9 9 9

tc 12 wred min = 28 22 16
tc 12 wred max = 32 32 32
tc 12 wred inv prob = 10 10 10
tc 12 wred weight = 9 9 9
```

With this configuration file, the RED configuration that applies to green, yellow and red packets in traffic class 0 is shown in Table 5.113.

Table 5.113: RED Configuration Corresponding to RED Configuration File

RED Parameter	Configuration Name	Green	Yellow	Red
Minimum Threshold	tc 0 wred min	28	22	16
Maximum Threshold	tc 0 wred max	32	32	32
Mark Probability	tc 0 wred inv prob	10	10	10
EWMA Filter Weight	tc 0 wred weight	9	9	9

Application Programming Interface (API)

Enqueue API

The syntax of the enqueue API is as follows:

```
int rte_red_enqueue(const struct rte_red_config *red_cfg, struct rte_red *red, const unsigned␣
↪q, const uint64_t time)
```

The arguments passed to the enqueue API are configuration data, run-time data, the current size of the packet queue (in packets) and a value representing the current time. The time reference is in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium (see Section 26.2.4.5.1 “Internal Time Reference”). The dropper reuses the scheduler time stamps for performance reasons.

Empty API

The syntax of the empty API is as follows:

```
void rte_red_mark_queue_empty(struct rte_red *red, const uint64_t time)
```

The arguments passed to the empty API are run-time data and the current time in bytes.

5.43.4 Traffic Metering

The traffic metering component implements the Single Rate Three Color Marker (srTCM) and Two Rate Three Color Marker (trTCM) algorithms, as defined by IETF RFC 2697 and 2698 respectively. These algorithms meter the stream of incoming packets based on the allowance defined in advance for each traffic flow. As result, each incoming packet is tagged as green, yellow or red based on the monitored consumption of the flow the packet belongs to.

Functional Overview

The srTCM algorithm defines two token buckets for each traffic flow, with the two buckets sharing the same token update rate:

- Committed (C) bucket: fed with tokens at the rate defined by the Committed Information Rate (CIR) parameter (measured in IP packet bytes per second). The size of the C bucket is defined by the Committed Burst Size (CBS) parameter (measured in bytes);
- Excess (E) bucket: fed with tokens at the same rate as the C bucket. The size of the E bucket is defined by the Excess Burst Size (EBS) parameter (measured in bytes).

The trTCM algorithm defines two token buckets for each traffic flow, with the two buckets being updated with tokens at independent rates:

- Committed (C) bucket: fed with tokens at the rate defined by the Committed Information Rate (CIR) parameter (measured in bytes of IP packet per second). The size of the C bucket is defined by the Committed Burst Size (CBS) parameter (measured in bytes);
- Peak (P) bucket: fed with tokens at the rate defined by the Peak Information Rate (PIR) parameter (measured in IP packet bytes per second). The size of the P bucket is defined by the Peak Burst Size (PBS) parameter (measured in bytes).

Please refer to RFC 2697 (for srTCM) and RFC 2698 (for trTCM) for details on how tokens are consumed from the buckets and how the packet color is determined.

Color Blind and Color Aware Modes

For both algorithms, the color blind mode is functionally equivalent to the color aware mode with input color set as green. For color aware mode, a packet with red input color can only get the red output color, while a packet with yellow input color can only get the yellow or red output colors.

The reason why the color blind mode is still implemented distinctly than the color aware mode is that color blind mode can be implemented with fewer operations than the color aware mode.

Implementation Overview

For each input packet, the steps for the srTCM / trTCM algorithms are:

- Update the C and E / P token buckets. This is done by reading the current time (from the CPU timestamp counter), identifying the amount of time since the last bucket update and computing the associated number of tokens (according to the pre-configured bucket rate). The number of tokens in the bucket is limited by the pre-configured bucket size;
- Identify the output color for the current packet based on the size of the IP packet and the amount of tokens currently available in the C and E / P buckets; for color aware mode only, the input color of the packet is also considered. When the output color is not red, a number of tokens equal to the length of the IP packet are subtracted from the C or E /P or both buckets, depending on the algorithm and the output color of the packet.

5.44 Power Management

The DPDK Power Management feature allows users space applications to save power by dynamically adjusting CPU frequency or entering into different C-States.

- Adjusting the CPU frequency dynamically according to the utilization of RX queue.
- Entering into different deeper C-States according to the adaptive algorithms to speculate brief periods of time suspending the application if no packets are received.

The interfaces for adjusting the operating CPU frequency are in the power management library. C-State control is implemented in applications according to the different use cases.

5.44.1 CPU Frequency Scaling

The Linux kernel provides a cpufreq module for CPU frequency scaling for each lcore. For example, for cpuX, /sys/devices/system/cpu/cpuX/cpufreq/ has the following sys files for frequency scaling:

- affected_cpus
- bios_limit
- cpuinfo_cur_freq
- cpuinfo_max_freq
- cpuinfo_min_freq
- cpuinfo_transition_latency
- related_cpus
- scaling_available_frequencies
- scaling_available_governors
- scaling_cur_freq
- scaling_driver
- scaling_governor
- scaling_max_freq
- scaling_min_freq
- scaling_setspeed

In the DPDK, scaling_governor is configured in user space. Then, a user space application can prompt the kernel by writing scaling_setspeed to adjust the CPU frequency according to the strategies defined by the user space application.

5.44.2 Core-load Throttling through C-States

Core state can be altered by speculative sleeps whenever the specified lcore has nothing to do. In the DPDK, if no packet is received after polling, speculative sleeps can be triggered according the strategies defined by the user space application.

5.44.3 Per-core Turbo Boost

Individual cores can be allowed to enter a Turbo Boost state on a per-core basis. This is achieved by enabling Turbo Boost Technology in the BIOS, then looping through the relevant cores and enabling/disabling Turbo Boost on each core.

5.44.4 Use of Power Library in a Hyper-Threaded Environment

In the case where the power library is in use on a system with Hyper-Threading enabled, the frequency on the physical core is set to the highest frequency of the Hyper-Thread siblings. So even though an application may request a scale down, the core frequency will remain at the highest frequency until all Hyper-Threads on that core request a scale down.

5.44.5 API Overview of the Power Library

The main methods exported by power library are for CPU frequency scaling and include the following:

- **Freq up**: Prompt the kernel to scale up the frequency of the specific lcore.
- **Freq down**: Prompt the kernel to scale down the frequency of the specific lcore.
- **Freq max**: Prompt the kernel to scale up the frequency of the specific lcore to the maximum.
- **Freq min**: Prompt the kernel to scale down the frequency of the specific lcore to the minimum.
- **Get available freqs**: Read the available frequencies of the specific lcore from the sys file.
- **Freq get**: Get the current frequency of the specific lcore.
- **Freq set**: Prompt the kernel to set the frequency for the specific lcore.
- **Enable turbo**: Prompt the kernel to enable Turbo Boost for the specific lcore.
- **Disable turbo**: Prompt the kernel to disable Turbo Boost for the specific lcore.

5.44.6 User Cases

The power management mechanism is used to save power when performing L3 forwarding.

5.44.7 Empty Poll API

Abstract

For packet processing workloads such as DPDK polling is continuous. This means CPU cores always show 100% busy independent of how much work those cores are doing. It is critical to accurately determine how busy a core is hugely important for the following reasons:

- No indication of overload conditions
- User does not know how much real load is on a system, resulting in wasted energy as no power management is utilized

Compared to the original l3fwd-power design, instead of going to sleep after detecting an empty poll, the new mechanism just lowers the core frequency. As a result, the application does not stop polling the device, which leads to improved handling of bursts of traffic.

When the system become busy, the empty poll mechanism can also increase the core frequency (including turbo) to do best effort for intensive traffic. This gives us more flexible and balanced traffic awareness over the standard l3fwd-power application.

Proposed Solution

The proposed solution focuses on how many times empty polls are executed. The less the number of empty polls, means current core is busy with processing workload, therefore, the higher frequency is needed. The high empty poll number indicates the current core not doing any real work therefore, we can lower the frequency to safe power.

In the current implementation, each core has 1 empty-poll counter which assume 1 core is dedicated to 1 queue. This will need to be expanded in the future to support multiple queues per core.

Power state definition:

- LOW: Not currently used, reserved for future use.
- MED: the frequency is used to process modest traffic workload.
- HIGH: the frequency is used to process busy traffic workload.

There are two phases to establish the power management system:

- Training phase. This phase is used to measure the optimal frequency change thresholds for a given system. The thresholds will differ from system to system due to differences in processor micro-architecture, cache and device configurations. In this phase, the user must ensure that no traffic can enter the system so that counts can be measured for empty polls at low, medium and high frequencies. Each frequency is measured for two seconds. Once the training phase is complete, the threshold numbers are displayed, and normal mode resumes, and traffic can be allowed into the system. These threshold number can be used on the command line when starting the application in normal mode to avoid re-training every time.
- Normal phase. Every 10ms the run-time counters are compared to the supplied threshold values, and the decision will be made whether to move to a different power state (by adjusting the frequency).

API Overview for Empty Poll Power Management

- **State Init**: initialize the power management system.
- **State Free**: free the resource hold by power management system.
- **Update Empty Poll Counter**: update the empty poll counter.
- **Update Valid Poll Counter**: update the valid poll counter.
- **Set the Frequency Index**: update the power state/frequency mapping.
- **Detect empty poll state change**: empty poll state change detection algorithm then take action.

5.44.8 User Cases

The mechanism can applied to any device which is based on polling. e.g. NIC, FPGA.

5.44.9 References

- The *L3 Forwarding with Power Management Sample Application* chapter in the *Sample Applications User Guides* section.
- The *Virtual Machine Power Management Application* chapter in the *Sample Applications User Guides* section.

5.45 Packet Classification and Access Control

The DPDK provides an Access Control library that gives the ability to classify an input packet based on a set of classification rules.

The ACL library is used to perform an N-tuple search over a set of rules with multiple categories and find the best match (highest priority) for each category. The library API provides the following basic operations:

- Create a new Access Control (AC) context.
- Add rules into the context.
- For all rules in the context, build the runtime structures necessary to perform packet classification.
- Perform input packet classifications.
- Destroy an AC context and its runtime structures and free the associated memory.

5.45.1 Overview

Rule definition

The current implementation allows the user for each AC context to specify its own rule (set of fields) over which packet classification will be performed. Though there are few restrictions on the rule fields layout:

- First field in the rule definition has to be one byte long.
- All subsequent fields has to be grouped into sets of 4 consecutive bytes.

This is done mainly for performance reasons - search function processes the first input byte as part of the flow setup and then the inner loop of the search function is unrolled to process four input bytes at a time.

To define each field inside an AC rule, the following structure is used:

```
struct rte_acl_field_def {
    uint8_t type;         /*< type - ACL_FIELD_TYPE. */
    uint8_t size;         /*< size of field 1,2,4, or 8. */
    uint8_t field_index;  /*< index of field inside the rule. */
    uint8_t input_index;  /*< 0-N input index. */
    uint32_t offset;      /*< offset to start of field. */
};
```

- type The field type is one of three choices:
  - _MASK - for fields such as IP addresses that have a value and a mask defining the number of relevant bits.
  - _RANGE - for fields such as ports that have a lower and upper value for the field.
  - _BITMASK - for fields such as protocol identifiers that have a value and a bit mask.
- size The size parameter defines the length of the field in bytes. Allowable values are 1, 2, 4, or 8 bytes. Note that due to the grouping of input bytes, 1 or 2 byte fields must be defined as consecutive fields that make up 4 consecutive input bytes. Also, it is best to define fields of 8 or more bytes as 4 byte fields so that the build processes can eliminate fields that are all wild.
- field_index A zero-based value that represents the position of the field inside the rule; 0 to N-1 for N fields.
- input_index As mentioned above, all input fields, except the very first one, must be in groups of 4 consecutive bytes. The input index specifies to which input group that field belongs to.
- offset The offset field defines the offset for the field. This is the offset from the beginning of the buffer parameter for the search.

For example, to define classification for the following IPv4 5-tuple structure:

```
struct ipv4_5tuple {
    uint8_t proto;
    uint32_t ip_src;
    uint32_t ip_dst;
    uint16_t port_src;
    uint16_t port_dst;
};
```

The following array of field definitions can be used:

```
struct rte_acl_field_def ipv4_defs[5] = {
    /* first input field - always one byte long. */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct ipv4_5tuple, proto),
    },

    /* next input field (IPv4 source address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
       .offset = offsetof (struct ipv4_5tuple, ip_src),
    },

    /* next input field (IPv4 destination address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 2,
        .input_index = 2,
       .offset = offsetof (struct ipv4_5tuple, ip_dst),
    },

    /*
     * Next 2 fields (src & dst ports) form 4 consecutive bytes.
     * They share the same input index.
     */
    {
        .type = RTE_ACL_FIELD_TYPE_RANGE,
        .size = sizeof (uint16_t),
        .field_index = 3,
        .input_index = 3,
        .offset = offsetof (struct ipv4_5tuple, port_src),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_RANGE,
        .size = sizeof (uint16_t),
        .field_index = 4,
        .input_index = 3,
        .offset = offsetof (struct ipv4_5tuple, port_dst),
    },
};
```

A typical example of such an IPv4 5-tuple rule is a follows:

```
source addr/mask  destination addr/mask  source ports dest ports protocol/mask
192.168.1.0/24    192.168.2.31/32        0:65535      1234:1234  17/0xff
```

Any IPv4 packets with protocol ID 17 (UDP), source address 192.168.1.[0-255], destination address 192.168.2.31, source port [0-65535] and destination port 1234 matches the above rule.

To define classification for the IPv6 2-tuple: <protocol, IPv6 source address> over the following IPv6 header structure:

```
struct rte_ipv6_hdr {
    uint32_t vtc_flow;     /* IP version, traffic class & flow label. */
    uint16_t payload_len;  /* IP packet length - includes sizeof(ip_header). */
    uint8_t proto;         /* Protocol, next header. */
    uint8_t hop_limits;    /* Hop limits. */
    uint8_t src_addr[16];  /* IP address of source host. */
    uint8_t dst_addr[16];  /* IP address of destination host(s). */
} __rte_packed;
```

The following array of field definitions can be used:

```
struct rte_acl_field_def ipv6_2tuple_defs[5] = {
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct rte_ipv6_hdr, proto),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
        .offset = offsetof (struct rte_ipv6_hdr, src_addr[0]),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 2,
        .input_index = 2,
        .offset = offsetof (struct rte_ipv6_hdr, src_addr[4]),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 3,
        .input_index = 3,
       .offset = offsetof (struct rte_ipv6_hdr, src_addr[8]),
    },

    {
       .type = RTE_ACL_FIELD_TYPE_MASK,
       .size = sizeof (uint32_t),
       .field_index = 4,
       .input_index = 4,
       .offset = offsetof (struct rte_ipv6_hdr, src_addr[12]),
    },
};
```

A typical example of such an IPv6 2-tuple rule is a follows:

```
source addr/mask                            protocol/mask
2001:db8:1234:0000:0000:0000:0000:0000/48    6/0xff
```

Any IPv6 packets with protocol ID 6 (TCP), and source address inside the range [2001:db8:1234:0000:0000:0000:0000:0000 - 2001:db8:1234:ffff:ffff:ffff:ffff:ffff] matches the above rule.

In the following example the last element of the search key is 8-bit long. So it is a case where the 4 consecutive bytes of an input field are not fully occupied. The structure for the classification is:

```
struct acl_key {
    uint8_t ip_proto;
    uint32_t ip_src;
    uint32_t ip_dst;
    uint8_t tos;      /*< This is partially using a 32-bit input element */
};
```

The following array of field definitions can be used:

```
struct rte_acl_field_def ipv4_defs[4] = {
    /* first input field - always one byte long. */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct acl_key, ip_proto),
    },

    /* next input field (IPv4 source address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
       .offset = offsetof (struct acl_key, ip_src),
    },

    /* next input field (IPv4 destination address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 2,
        .input_index = 2,
       .offset = offsetof (struct acl_key, ip_dst),
    },

    /*
     * Next element of search key (Type of Service) is indeed 1 byte long.
     * Anyway we need to allocate all the 4 consecutive bytes for it.
     */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint32_t), /* All the 4 consecutive bytes are allocated */
        .field_index = 3,
        .input_index = 3,
        .offset = offsetof (struct acl_key, tos),
    },
};
```

A typical example of such an IPv4 4-tuple rule is as follows:

```
source addr/mask  destination addr/mask  tos/mask protocol/mask
192.168.1.0/24    192.168.2.31/32        1/0xff   6/0xff
```

Any IPv4 packets with protocol ID 6 (TCP), source address 192.168.1.[0-255], destination address 192.168.2.31, ToS 1 matches the above rule.

When creating a set of rules, for each rule, additional information must be supplied also:

- **priority**: A weight to measure the priority of the rules (higher is better). If the input tuple matches more than one rule, then the rule with the higher priority is returned. Note that if the input tuple matches more than one rule and these rules have equal priority, it is undefined which rule is returned as a match. It is recommended to assign a unique priority for each rule.
- **category_mask**: Each rule uses a bit mask value to select the relevant category(s) for the rule. When a lookup is performed, the result for each category is returned. This effectively provides a "parallel lookup" by enabling a single search to return multiple results if, for example, there were four different sets of ACL rules, one for access control, one for routing, and so on. Each set could be assigned its own category and by combining them into a single database, one lookup returns a result for each of the four sets.
- **userdata**: A user-defined value. For each category, a successful match returns the userdata field of the highest priority matched rule. When no rules match, returned value is zero.

Note: When adding new rules into an ACL context, all fields must be in host byte order (LSB). When the search is performed for an input tuple, all fields in that tuple must be in network byte order (MSB).

RT memory size limit

Build phase (rte_acl_build()) creates for a given set of rules internal structure for further run-time traversal. With current implementation it is a set of multi-bit tries (with stride == 8). Depending on the rules set, that could consume significant amount of memory. In attempt to conserve some space ACL build process tries to split the given rule-set into several non-intersecting subsets and construct a separate trie for each of them. Depending on the rule-set, it might reduce RT memory requirements but might increase classification time. There is a possibility at build-time to specify maximum memory limit for internal RT structures for given AC context. It could be done via **max_size** field of the **rte_acl_config** structure. Setting it to the value greater than zero, instructs rte_acl_build() to:

- attempt to minimize number of tries in the RT table, but
- make sure that size of RT table wouldn't exceed given value.

Setting it to zero makes rte_acl_build() to use the default behavior: try to minimize size of the RT structures, but doesn't expose any hard limit on it.

That gives the user the ability to decisions about performance/space trade-off. For example:

```
struct rte_acl_ctx * acx;
struct rte_acl_config cfg;
int ret;

/*
 * assuming that acx points to already created and
 * populated with rules AC context and cfg filled properly.
 */

 /* try to build AC context, with RT structures less then 8MB. */
 cfg.max_size = 0x800000;
 ret = rte_acl_build(acx, &cfg);

 /*
  * RT structures can't fit into 8MB for given context.
  * Try to build without exposing any hard limit.
  */
```

(continues on next page)

(continued from previous page)

```
if (ret == -ERANGE) {
   cfg.max_size = 0;
   ret = rte_acl_build(acx, &cfg);
}
```

Classification methods

After rte_acl_build() over given AC context has finished successfully, it can be used to perform classification - search for a rule with highest priority over the input data. There are several implementations of classify algorithm:

- **RTE_ACL_CLASSIFY_SCALAR**: generic implementation, doesn't require any specific HW support.
- **RTE_ACL_CLASSIFY_SSE**: vector implementation, can process up to 8 flows in parallel. Requires SSE 4.1 support.
- **RTE_ACL_CLASSIFY_AVX2**: vector implementation, can process up to 16 flows in parallel. Requires AVX2 support.

It is purely a runtime decision which method to choose, there is no build-time difference. All implementations operates over the same internal RT structures and use similar principles. The main difference is that vector implementations can manually exploit IA SIMD instructions and process several input data flows in parallel. At startup ACL library determines the highest available classify method for the given platform and sets it as default one. Though the user has an ability to override the default classifier function for a given ACL context or perform particular search using non-default classify method. In that case it is user responsibility to make sure that given platform supports selected classify implementation.

5.45.2 Application Programming Interface (API) Usage

Note: For more details about the Access Control API, please refer to the *DPDK API Reference*.

The following example demonstrates IPv4, 5-tuple classification for rules defined above with multiple categories in more detail.

Classify with Multiple Categories

```
struct rte_acl_ctx * acx;
struct rte_acl_config cfg;
int ret;

/* define a structure for the rule with up to 5 fields. */

RTE_ACL_RULE_DEF(acl_ipv4_rule, RTE_DIM(ipv4_defs));

/* AC context creation parameters. */

struct rte_acl_param prm = {
    .name = "ACL_example",
    .socket_id = SOCKET_ID_ANY,
```

(continues on next page)

(continued from previous page)

```
    .rule_size = RTE_ACL_RULE_SZ(RTE_DIM(ipv4_defs)),

    /* number of fields per rule. */

    .max_rule_num = 8, /* maximum number of rules in the AC context. */
};

struct acl_ipv4_rule acl_rules[] = {

    /* matches all packets traveling to 192.168.0.0/16, applies for categories: 0,1 */
    {
        .data = {.userdata = 1, .category_mask = 3, .priority = 1},

        /* destination IPv4 */
        .field[2] = {.value.u32 = RTE_IPV4(192,168,0,0),. mask_range.u32 = 16,},

        /* source port */
        .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

        /* destination port */
       .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
    },

    /* matches all packets traveling to 192.168.1.0/24, applies for categories: 0 */
    {
        .data = {.userdata = 2, .category_mask = 1, .priority = 2},

        /* destination IPv4 */
        .field[2] = {.value.u32 = RTE_IPV4(192,168,1,0),. mask_range.u32 = 24,},

        /* source port */
        .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

        /* destination port */
        .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
    },

    /* matches all packets traveling from 10.1.1.1, applies for categories: 1 */
    {
        .data = {.userdata = 3, .category_mask = 2, .priority = 3},

        /* source IPv4 */
        .field[1] = {.value.u32 = RTE_IPV4(10,1,1,1),. mask_range.u32 = 32,},

        /* source port */
        .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

        /* destination port */
        .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
    },

};


/* create an empty AC context  */

if ((acx = rte_acl_create(&prm)) == NULL) {

    /* handle context create failure. */

}
```

(continues on next page)

(continued from previous page)

```
/* add rules to the context */

ret = rte_acl_add_rules(acx, acl_rules, RTE_DIM(acl_rules));
if (ret != 0) {
   /* handle error at adding ACL rules. */
}

/* prepare AC build config. */

cfg.num_categories = 2;
cfg.num_fields = RTE_DIM(ipv4_defs);

memcpy(cfg.defs, ipv4_defs, sizeof (ipv4_defs));

/* build the runtime structures for added rules, with 2 categories. */

ret = rte_acl_build(acx, &cfg);
if (ret != 0) {
   /* handle error at build runtime structures for ACL context. */
}
```

For a tuple with source IP address: 10.1.1.1 and destination IP address: 192.168.1.15, once the following lines are executed:

```
uint32_t results[4]; /* make classify for 4 categories. */

rte_acl_classify(acx, data, results, 1, 4);
```

then the results[] array contains:

```
results[4] = {2, 3, 0, 0};
```

- For category 0, both rules 1 and 2 match, but rule 2 has higher priority, therefore results[0] contains the userdata for rule 2.
- For category 1, both rules 1 and 3 match, but rule 3 has higher priority, therefore results[1] contains the userdata for rule 3.
- For categories 2 and 3, there are no matches, so results[2] and results[3] contain zero, which indicates that no matches were found for those categories.

For a tuple with source IP address: 192.168.1.1 and destination IP address: 192.168.2.11, once the following lines are executed:

```
uint32_t results[4]; /* make classify by 4 categories. */

rte_acl_classify(acx, data, results, 1, 4);
```

the results[] array contains:

```
results[4] = {1, 1, 0, 0};
```

- For categories 0 and 1, only rule 1 matches.
- For categories 2 and 3, there are no matches.

For a tuple with source IP address: 10.1.1.1 and destination IP address: 201.212.111.12, once the following lines are executed:

```
uint32_t results[4]; /* make classify by 4 categories. */
rte_acl_classify(acx, data, results, 1, 4);
```

the results[] array contains:

```
results[4] = {0, 3, 0, 0};
```

- For category 1, only rule 3 matches.
- For categories 0, 2 and 3, there are no matches.

5.46 Packet Framework

5.46.1 Design Objectives

The main design objectives for the DPDK Packet Framework are:

- Provide standard methodology to build complex packet processing pipelines. Provide reusable and extensible templates for the commonly used pipeline functional blocks;
- Provide capability to switch between pure software and hardware-accelerated implementations for the same pipeline functional block;
- Provide the best trade-off between flexibility and performance. Hardcoded pipelines usually provide the best performance, but are not flexible, while developing flexible frameworks is never a problem, but performance is usually low;
- Provide a framework that is logically similar to Open Flow.

5.46.2 Overview

Packet processing applications are frequently structured as pipelines of multiple stages, with the logic of each stage glued around a lookup table. For each incoming packet, the table defines the set of actions to be applied to the packet, as well as the next stage to send the packet to.

The DPDK Packet Framework minimizes the development effort required to build packet processing pipelines by defining a standard methodology for pipeline development, as well as providing libraries of reusable templates for the commonly used pipeline blocks.

The pipeline is constructed by connecting the set of input ports with the set of output ports through the set of tables in a tree-like topology. As result of lookup operation for the current packet in the current table, one of the table entries (on lookup hit) or the default table entry (on lookup miss) provides the set of actions to be applied on the current packet, as well as the next hop for the packet, which can be either another table, an output port or packet drop.

An example of packet processing pipeline is presented in Fig. 5.79:

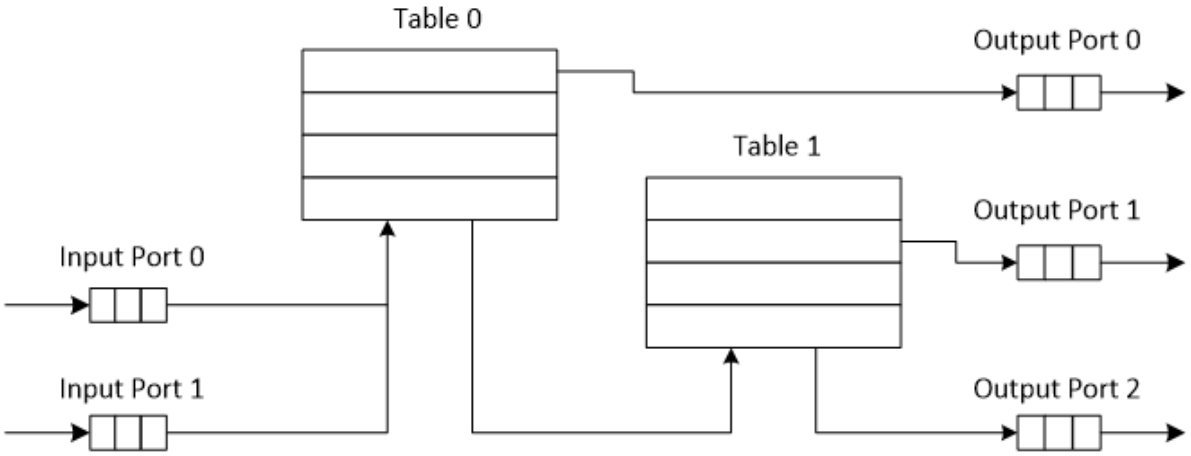


Fig. 5.79: Example of Packet Processing Pipeline where Input Ports 0 and 1 are Connected with Output Ports 0, 1 and 2 through Tables 0 and 1

5.46.3 Port Library Design

Port Types

Table 5.114 is a non-exhaustive list of ports that can be implemented with the Packet Framework.

Table 5.114: Port Types

#	Port type	Description
1	SW ring	SW circular buffer used for message passing between the application threads. Uses the DPDK rte_ring primitive. Expected to be the most commonly used type of port.
2	HW ring	Queue of buffer descriptors used to interact with NIC, switch or accelerator ports. For NIC ports, it uses the DPDK rte_eth_rx_queue or rte_eth_tx_queue primitives.
3	IP reassembly	Input packets are either IP fragments or complete IP datagrams. Output packets are complete IP datagrams.
4	IP fragmentation	Input packets are jumbo (IP datagrams with length bigger than MTU) or non-jumbo packets. Output packets are non-jumbo packets.
5	Traffic manager	Traffic manager attached to a specific NIC output port, performing congestion management and hierarchical scheduling according to pre-defined SLAs.
6	KNI	Send/receive packets to/from Linux kernel space.
7	Source	Input port used as packet generator. Similar to Linux kernel /dev/zero character device.
8	Sink	Output port used to drop all input packets. Similar to Linux kernel /dev/null character device.
9	Sym_crypto	Output port used to extract DPDK Cryptodev operations from a fixed offset of the packet and then enqueue to the Cryptodev PMD. Input port used to dequeue the Cryptodev operations from the Cryptodev PMD and then retrieve the packets from them.

Port Interface

Each port is unidirectional, i.e. either input port or output port. Each input/output port is required to implement an abstract interface that defines the initialization and run-time operation of the port. The port abstract interface is described in.

Table 5.115: 20 Port Abstract Interface

#	Port Operation	Description
1	Create	Create the low-level port object (e.g. queue). Can internally allocate memory.
2	Free	Free the resources (e.g. memory) used by the low-level port object.
3	RX	Read a burst of input packets. Non-blocking operation. Only defined for input ports.
4	TX	Write a burst of input packets. Non-blocking operation. Only defined for output ports.
5	Flush	Flush the output buffer. Only defined for output ports.

5.46.4 Table Library Design

Table Types

Table 5.116 is a non-exhaustive list of types of tables that can be implemented with the Packet Framework.

Table 5.116: Table Types

#	Table Type	Description
1	Hash table	Lookup key is n-tuple based. Typically, the lookup key is hashed to produce a signature that is used to identify a bucket of entries where the lookup key is searched next. The signature associated with the lookup key of each input packet is either read from the packet descriptor (pre-computed signature) or computed at table lookup time. The table lookup, add entry and delete entry operations, as well as any other pipeline block that pre-computes the signature all have to use the same hashing algorithm to generate the signature. Typically used to implement flow classification tables, ARP caches, routing table for tunnelling protocols, etc.
2	Longest Prefix Match (LPM)	Lookup key is the IP address. Each table entries has an associated IP prefix (IP and depth). The table lookup operation selects the IP prefix that is matched by the lookup key; in case of multiple matches, the entry with the longest prefix depth wins. Typically used to implement IP routing tables.
3	Access Control List (ACLs)	Lookup key is 7-tuple of two VLAN/MPLS labels, IP destination address, IP source addresses, L4 protocol, L4 destination port, L4 source port. Each table entry has an associated ACL and priority. The ACL contains bit masks for the VLAN/MPLS labels, IP prefix for IP destination address, IP prefix for IP source addresses, L4 protocol and bitmask, L4 destination port and bit mask, L4 source port and bit mask. The table lookup operation selects the ACL that is matched by the lookup key; in case of multiple matches, the entry with the highest priority wins. Typically used to implement rule databases for firewalls, etc.
4	Pattern matching search	Lookup key is the packet payload. Table is a database of patterns, with each pattern having a priority assigned. The table lookup operation selects the patterns that is matched by the input packet; in case of multiple matches, the matching pattern with the highest priority wins.
5	Array	Lookup key is the table entry index itself.

Table Interface

Each table is required to implement an abstract interface that defines the initialization and run-time operation of the table. The table abstract interface is described in Table 5.117.

Table 5.117: Table Abstract Interface

#	Table operation	Description
1	Create	Create the low-level data structures of the lookup table. Can internally allocate memory.
2	Free	Free up all the resources used by the lookup table.
3	Add entry	Add new entry to the lookup table.
4	Delete entry	Delete specific entry from the lookup table.
5	Lookup	Look up a burst of input packets and return a bit mask specifying the result of the lookup operation for each packet: a set bit signifies lookup hit for the corresponding packet, while a cleared bit a lookup miss. For each lookup hit packet, the lookup operation also returns a pointer to the table entry that was hit, which contains the actions to be applied on the packet and any associated metadata. For each lookup miss packet, the actions to be applied on the packet and any associated metadata are specified by the default table entry preconfigured for lookup miss.

Hash Table Design

Hash Table Overview

Hash tables are important because the key lookup operation is optimized for speed: instead of having to linearly search the lookup key through all the keys in the table, the search is limited to only the keys stored in a single table bucket.

Associative Arrays

An associative array is a function that can be specified as a set of (key, value) pairs, with each key from the possible set of input keys present at most once. For a given associative array, the possible operations are:

1. *add (key, value)*: When no value is currently associated with *key*, then the (key, *value*) association is created. When *key* is already associated value *value0*, then the association (*key*, *value0*) is removed and association *(key, value)* is created;

2. *delete key*: When no value is currently associated with *key*, this operation has no effect. When *key* is already associated *value*, then association *(key, value)* is removed;

3. *lookup key*: When no value is currently associated with *key*, then this operation returns void value (lookup miss). When *key* is associated with *value*, then this operation returns *value*. The *(key, value)* association is not changed.

The matching criterion used to compare the input key against the keys in the associative array is *exact match*, as the key size (number of bytes) and the key value (array of bytes) have to match exactly for the

two keys under comparison.

Hash Function

A hash function deterministically maps data of variable length (key) to data of fixed size (hash value or key signature). Typically, the size of the key is bigger than the size of the key signature. The hash function basically compresses a long key into a short signature. Several keys can share the same signature (collisions).

High quality hash functions have uniform distribution. For large number of keys, when dividing the space of signature values into a fixed number of equal intervals (buckets), it is desirable to have the key signatures evenly distributed across these intervals (uniform distribution), as opposed to most of the signatures going into only a few of the intervals and the rest of the intervals being largely unused (non-uniform distribution).

Hash Table

A hash table is an associative array that uses a hash function for its operation. The reason for using a hash function is to optimize the performance of the lookup operation by minimizing the number of table keys that have to be compared against the input key.

Instead of storing the (key, value) pairs in a single list, the hash table maintains multiple lists (buckets). For any given key, there is a single bucket where that key might exist, and this bucket is uniquely identified based on the key signature. Once the key signature is computed and the hash table bucket identified, the key is either located in this bucket or it is not present in the hash table at all, so the key search can be narrowed down from the full set of keys currently in the table to just the set of keys currently in the identified table bucket.

The performance of the hash table lookup operation is greatly improved, provided that the table keys are evenly distributed among the hash table buckets, which can be achieved by using a hash function with uniform distribution. The rule to map a key to its bucket can simply be to use the key signature (modulo the number of table buckets) as the table bucket ID:

bucket_id = f_hash(key) % n_buckets;

By selecting the number of buckets to be a power of two, the modulo operator can be replaced by a bitwise AND logical operation:

bucket_id = f_hash(key) & (n_buckets - 1);

considering *n_bits* as the number of bits set in *bucket_mask = n_buckets - 1*, this means that all the keys that end up in the same hash table bucket have the lower *n_bits* of their signature identical. In order to reduce the number of keys in the same bucket (collisions), the number of hash table buckets needs to be increased.

In packet processing context, the sequence of operations involved in hash table operations is described in Fig. 5.80:

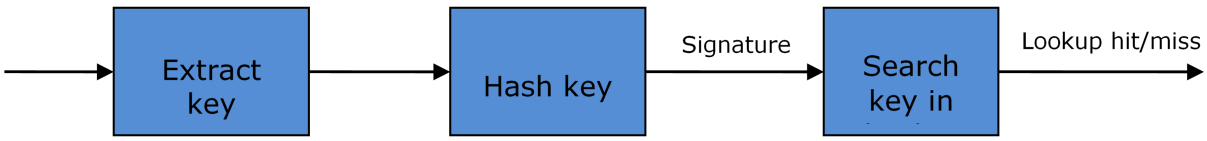


Fig. 5.80: Sequence of Steps for Hash Table Operations in a Packet Processing Context

Hash Table Use Cases

Flow Classification

Description: The flow classification is executed at least once for each input packet. This operation maps each incoming packet against one of the known traffic flows in the flow database that typically contains millions of flows.

Hash table name: Flow classification table

Number of keys: Millions

Key format: n-tuple of packet fields that uniquely identify a traffic flow/connection. Example: DiffServ 5-tuple of (Source IP address, Destination IP address, L4 protocol, L4 protocol source port, L4 protocol destination port). For IPv4 protocol and L4 protocols like TCP, UDP or SCTP, the size of the DiffServ 5-tuple is 13 bytes, while for IPv6 it is 37 bytes.

Key value (key data): actions and action meta-data describing what processing to be applied for the packets of the current flow. The size of the data associated with each traffic flow can vary from 8 bytes to kilobytes.

Address Resolution Protocol (ARP)

Description: Once a route has been identified for an IP packet (so the output interface and the IP address of the next hop station are known), the MAC address of the next hop station is needed in order to send this packet onto the next leg of the journey towards its destination (as identified by its destination IP address). The MAC address of the next hop station becomes the destination MAC address of the outgoing Ethernet frame.

Hash table name: ARP table

Number of keys: Thousands

Key format: The pair of (Output interface, Next Hop IP address), which is typically 5 bytes for IPv4 and 17 bytes for IPv6.

Key value (key data): MAC address of the next hop station (6 bytes).

Hash Table Types

Table 5.118 lists the hash table configuration parameters shared by all different hash table types.

Table 5.118: Configuration Parameters Common for All Hash Table Types

#	Parameter	Details
1	Key size	Measured as number of bytes. All keys have the same size.
2	Key value (key data) size	Measured as number of bytes.
3	Number of buckets	Needs to be a power of two.
4	Maximum number of keys	Needs to be a power of two.
5	Hash function	Examples: jhash, CRC hash, etc.
6	Hash function seed	Parameter to be passed to the hash function.
7	Key offset	Offset of the lookup key byte array within the packet meta-data stored in the packet buffer.

Bucket Full Problem

On initialization, each hash table bucket is allocated space for exactly 4 keys. As keys are added to the table, it can happen that a given bucket already has 4 keys when a new key has to be added to this bucket. The possible options are:

1. **Least Recently Used (LRU) Hash Table.** One of the existing keys in the bucket is deleted and the new key is added in its place. The number of keys in each bucket never grows bigger than 4. The logic to pick the key to be dropped from the bucket is LRU. The hash table lookup operation maintains the order in which the keys in the same bucket are hit, so every time a key is hit, it becomes the new Most Recently Used (MRU) key, i.e. the last candidate for drop. When a key is added to the bucket, it also becomes the new MRU key. When a key needs to be picked and dropped, the first candidate for drop, i.e. the current LRU key, is always picked. The LRU logic requires maintaining specific data structures per each bucket.
2. **Extendable Bucket Hash Table.** The bucket is extended with space for 4 more keys. This is done by allocating additional memory at table initialization time, which is used to create a pool of free keys (the size of this pool is configurable and always a multiple of 4). On key add operation, the allocation of a group of 4 keys only happens successfully within the limit of free keys, otherwise the key add operation fails. On key delete operation, a group of 4 keys is freed back to the pool of free keys when the key to be deleted is the only key that was used within its group of 4 keys at that time. On key lookup operation, if the current bucket is in extended state and a match is not found in the first group of 4 keys, the search continues beyond the first group of 4 keys, potentially until all keys in this bucket are examined. The extendable bucket logic requires maintaining specific data structures per table and per each bucket.

Table 5.119: Configuration Parameters Specific to Extendable Bucket Hash Table

#	Parameter	Details
1	Number of additional keys	Needs to be a power of two, at least equal to 4.

Signature Computation

The possible options for key signature computation are:

1. **Pre-computed key signature.** The key lookup operation is split between two CPU cores. The first CPU core (typically the CPU core that performs packet RX) extracts the key from the input packet, computes the key signature and saves both the key and the key signature in the packet buffer as packet meta-data. The second CPU core reads both the key and the key signature from the packet meta-data and performs the bucket search step of the key lookup operation.
2. **Key signature computed on lookup ("do-sig" version).** The same CPU core reads the key from the packet meta-data, uses it to compute the key signature and also performs the bucket search step of the key lookup operation.

Table 5.120: Configuration Parameters Specific to Pre-computed Key Signature Hash Table

#	Parameter	Details
1	Signature offset	Offset of the pre-computed key signature within the packet meta-data.

Key Size Optimized Hash Tables

For specific key sizes, the data structures and algorithm of key lookup operation can be specially hand-crafted for further performance improvements, so following options are possible:

1. **Implementation supporting configurable key size.**
2. **Implementation supporting a single key size.** Typical key sizes are 8 bytes and 16 bytes.

Bucket Search Logic for Configurable Key Size Hash Tables

The performance of the bucket search logic is one of the main factors influencing the performance of the key lookup operation. The data structures and algorithm are designed to make the best use of Intel CPU architecture resources like: cache memory space, cache memory bandwidth, external memory bandwidth, multiple execution units working in parallel, out of order instruction execution, special CPU instructions, etc.

The bucket search logic handles multiple input packets in parallel. It is built as a pipeline of several stages (3 or 4), with each pipeline stage handling two different packets from the burst of input packets. On each pipeline iteration, the packets are pushed to the next pipeline stage: for the 4-stage pipeline, two packets (that just completed stage 3) exit the pipeline, two packets (that just completed stage 2) are now executing stage 3, two packets (that just completed stage 1) are now executing stage 2, two packets (that just completed stage 0) are now executing stage 1 and two packets (next two packets to read from the burst of input packets) are entering the pipeline to execute stage 0. The pipeline iterations continue until all packets from the burst of input packets execute the last stage of the pipeline.

The bucket search logic is broken into pipeline stages at the boundary of the next memory access. Each pipeline stage uses data structures that are stored (with high probability) into the L1 or L2 cache memory of the current CPU core and breaks just before the next memory access required by the algorithm. The current pipeline stage finalizes by prefetching the data structures required by the next pipeline stage, so given enough time for the prefetch to complete, when the next pipeline stage eventually gets executed for the same packets, it will read the data structures it needs from L1 or L2 cache memory and thus avoid the significant penalty incurred by L2 or L3 cache memory miss.

By prefetching the data structures required by the next pipeline stage in advance (before they are used) and switching to executing another pipeline stage for different packets, the number of L2 or L3 cache memory misses is greatly reduced, hence one of the main reasons for improved performance. This is because the cost of L2/L3 cache memory miss on memory read accesses is high, as usually due to data dependency between instructions, the CPU execution units have to stall until the read operation is completed from L3 cache memory or external DRAM memory. By using prefetch instructions, the latency of memory read accesses is hidden, provided that it is preformed early enough before the respective data structure is actually used.

By splitting the processing into several stages that are executed on different packets (the packets from the input burst are interlaced), enough work is created to allow the prefetch instructions to complete successfully (before the prefetched data structures are actually accessed) and also the data dependency between instructions is loosened. For example, for the 4-stage pipeline, stage 0 is executed on packets 0 and 1 and then, before same packets 0 and 1 are used (i.e. before stage 1 is executed on packets 0 and 1), different packets are used: packets 2 and 3 (executing stage 1), packets 4 and 5 (executing stage 2) and packets 6 and 7 (executing stage 3). By executing useful work while the data structures are brought into the L1 or L2 cache memory, the latency of the read memory accesses is hidden. By increasing the gap between two consecutive accesses to the same data structure, the data dependency between instructions is loosened; this allows making the best use of the super-scalar and out-of-order

execution CPU architecture, as the number of CPU core execution units that are active (rather than idle or stalled due to data dependency constraints between instructions) is maximized.

The bucket search logic is also implemented without using any branch instructions. This avoids the important cost associated with flushing the CPU core execution pipeline on every instance of branch misprediction.

Configurable Key Size Hash Table

Fig. 5.81, Table 5.121 and Table 5.122 detail the main data structures used to implement configurable key size hash tables (either LRU or extendable bucket, either with pre-computed signature or "do-sig").

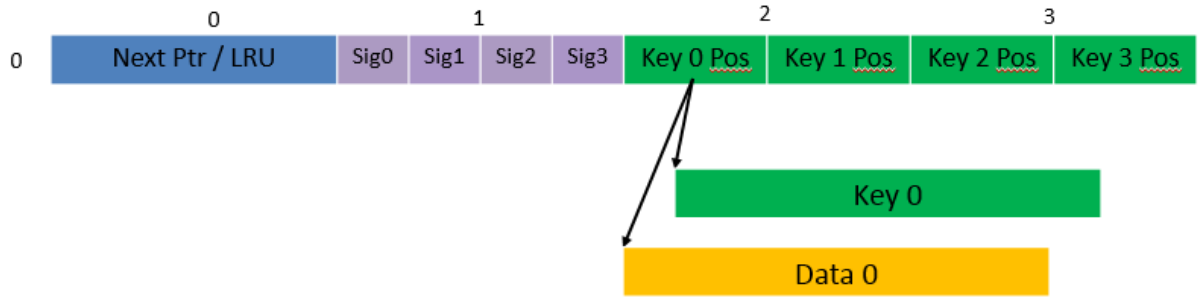


Fig. 5.81: Data Structures for Configurable Key Size Hash Tables

Table 5.121: Main Large Data Structures (Arrays) used for Configurable Key Size Hash Tables

#	Array name	Number of entries	Entry size (bytes)	Description
1	Bucket array	n_buckets (configurable)	32	Buckets of the hash table.
2	Bucket extensions array	n_buckets_ext (configurable)	32	This array is only created for extendable bucket tables.
3	Key array	n_keys	key_size (configurable)	Keys added to the hash table.
4	Data array	n_keys	entry_size (configurable)	Key values (key data) associated with the hash table keys.

Table 5.122: Field Description for Bucket Array Entry (Configurable Key Size Hash Tables)

#	Field name	Field size (bytes)	Description
1	Next Ptr/LRU	8	For LRU tables, this fields represents the LRU list for the current bucket stored as array of 4 entries of 2 bytes each. Entry 0 stores the index (0 .. 3) of the MRU key, while entry 3 stores the index of the LRU key. For extendable bucket tables, this field represents the next pointer (i.e. the pointer to the next group of 4 keys linked to the current bucket). The next pointer is not NULL if the bucket is currently extended or NULL otherwise. To help the branchless implementation, bit 0 (least significant bit) of this field is set to 1 if the next pointer is not NULL and to 0 otherwise.
2	Sig[0 .. 3]	4 x 2	If key X (X = 0 .. 3) is valid, then sig X bits 15 .. 1 store the most significant 15 bits of key X signature and sig X bit 0 is set to 1. If key X is not valid, then sig X is set to zero.
3	Key Pos [0 .. 3]	4 x 4	If key X is valid (X = 0 .. 3), then Key Pos X represents the index into the key array where key X is stored, as well as the index into the data array where the value associated with key X is stored. If key X is not valid, then the value of Key Pos X is undefined.

Fig. 5.82 and Table 5.123 detail the bucket search pipeline stages (either LRU or extendable bucket, either with pre-computed signature or "do-sig"). For each pipeline stage, the described operations are applied to each of the two packets handled by that stage.

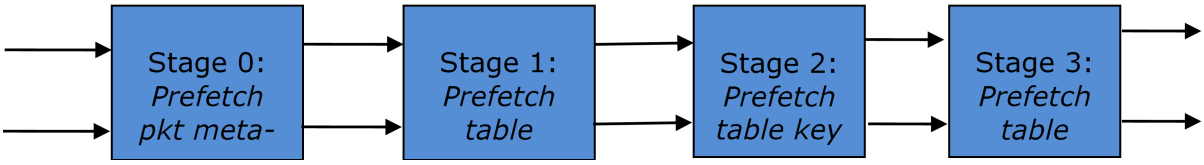


Fig. 5.82: Bucket Search Pipeline for Key Lookup Operation (Configurable Key Size Hash Tables)

Table 5.123: Description of the Bucket Search Pipeline Stages (Configurable Key Size Hash Tables)

#	Stage name	Description
0	Prefetch packet meta-data	Select next two packets from the burst of input packets. Prefetch packet meta-data containing the key and key signature.
1	Prefetch table bucket	Read the key signature from the packet meta-data (for extendable bucket hash tables) or read the key from the packet meta-data and compute key signature (for LRU tables). Identify the bucket ID using the key signature. Set bit 0 of the signature to 1 (to match only signatures of valid keys from the table). Prefetch the bucket.
2	Prefetch table key	Read the key signatures from the bucket. Compare the signature of the input key against the 4 key signatures from the packet. As result, the following is obtained: *match* = equal to TRUE if there was at least one signature match and to FALSE in the case of no signature match; *match_many* = equal to TRUE is there were more than one signature matches (can be up to 4 signature matches in the worst case scenario) and to FALSE otherwise; *match_pos* = the index of the first key that produced signature match (only valid if match is true). For extendable bucket hash tables only, set *match_many* to TRUE if next pointer is valid. Prefetch the bucket key indicated by *match_pos* (even if *match_pos* does not point to valid key valid).
3	Prefetch table data	Read the bucket key indicated by *match_pos*. Compare the bucket key against the input key. As result, the following is obtained: *match_key* = equal to TRUE if the two keys match and to FALSE otherwise. Report input key as lookup hit only when both *match* and *match_key* are equal to TRUE and as lookup miss otherwise. For LRU tables only, use branchless logic to update the bucket LRU list (the current key becomes the new MRU) only on lookup hit. Prefetch the key value (key data) associated with the current key (to avoid branches, this is done on both lookup hit and miss).

Additional notes:

1. The pipelined version of the bucket search algorithm is executed only if there are at least 7 packets in the burst of input packets. If there are less than 7 packets in the burst of input packets, a non-optimized implementation of the bucket search algorithm is executed.

2. Once the pipelined version of the bucket search algorithm has been executed for all the packets in the burst of input packets, the non-optimized implementation of the bucket search algorithm is also executed for any packets that did not produce a lookup hit, but have the *match_many* flag set. As result of executing the non-optimized version, some of these packets may produce a lookup hit or lookup miss. This does not impact the performance of the key lookup operation, as the probability of matching more than one signature in the same group of 4 keys or of having the bucket in extended state (for extendable bucket hash tables only) is relatively small.

Key Signature Comparison Logic

The key signature comparison logic is described in Table 5.124.

Table 5.124: Lookup Tables for Match, Match_Many and Match_Pos

#	mask	match (1 bit)	match_many (1 bit)	match_pos (2 bits)
0	0000	0	0	00
1	0001	1	0	00
2	0010	1	0	01
3	0011	1	1	00
4	0100	1	0	10
5	0101	1	1	00
6	0110	1	1	01
7	0111	1	1	00
8	1000	1	0	11
9	1001	1	1	00
10	1010	1	1	01
11	1011	1	1	00
12	1100	1	1	10
13	1101	1	1	00
14	1110	1	1	01
15	1111	1	1	00

The input *mask* hash bit X (X = 0 .. 3) set to 1 if input signature is equal to bucket signature X and set to 0 otherwise. The outputs *match*, *match_many* and *match_pos* are 1 bit, 1 bit and 2 bits in size respectively and their meaning has been explained above.

As displayed in Table 5.125, the lookup tables for *match* and *match_many* can be collapsed into a single 32-bit value and the lookup table for *match_pos* can be collapsed into a 64-bit value. Given the input *mask*, the values for *match*, *match_many* and *match_pos* can be obtained by indexing their respective bit array to extract 1 bit, 1 bit and 2 bits respectively with branchless logic.

Table 5.125: Collapsed Lookup Tables for Match, Match_Many and Match_Pos

	Bit array	Hexadecimal value
match	1111_1111_1111_1110	0xFFFELLU
match_many	1111_1110_1110_1000	0xFEE8LLU
match_pos	0001_0010_0001_0011__0001_0010_0001_0000	0x12131210LLU

The pseudo-code for match, match_many and match_pos is:

```
match = (0xFFFELLU >> mask) & 1;

match_many = (0xFEE8LLU >> mask) & 1;

match_pos = (0x12131210LLU >> (mask << 1)) & 3;
```

Single Key Size Hash Tables

Fig. 5.83, Fig. 5.84, Table 5.126 and Table 5.127 detail the main data structures used to implement 8-byte and 16-byte key hash tables (either LRU or extendable bucket, either with pre-computed signature or "do-sig").

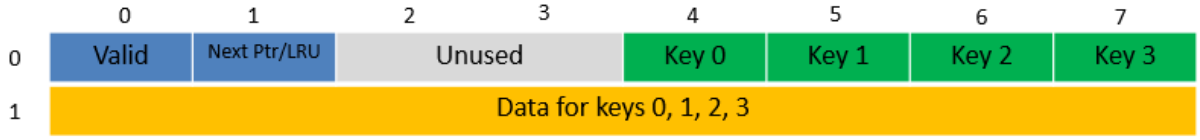


Fig. 5.83: Data Structures for 8-byte Key Hash Tables

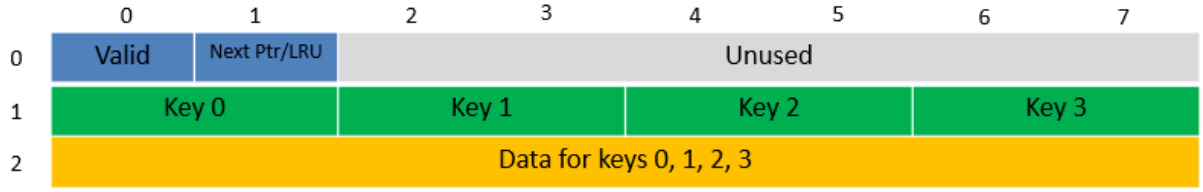


Fig. 5.84: Data Structures for 16-byte Key Hash Tables

Table 5.126: Main Large Data Structures (Arrays) used for 8-byte and 16-byte Key Size Hash Tables

#	Array name	Number of entries	Entry size (bytes)	Description
1	Bucket array	n_buckets (configurable)	*8-byte key size:* 64 + 4 x entry_size *16-byte key size:* 128 + 4 x entry_size	Buckets of the hash table.
2	Bucket extensions array	n_buckets_ext (configurable)	*8-byte key size:* 64 + 4 x entry_size *16-byte key size:* 128 + 4 x entry_size	This array is only created for extendable bucket tables.

Table 5.127: Field Description for Bucket Array Entry (8-byte and 16-byte Key Hash Tables)

#	Field name	Field size (bytes)	Description
1	Valid	8	Bit X (X = 0 .. 3) is set to 1 if key X is valid or to 0 otherwise. Bit 4 is only used for extendable bucket tables to help with the implementation of the branchless logic. In this case, bit 4 is set to 1 if next pointer is valid (not NULL) or to 0 otherwise.
2	Next Ptr/LRU	8	For LRU tables, this fields represents the LRU list for the current bucket stored as array of 4 entries of 2 bytes each. Entry 0 stores the index (0 .. 3) of the MRU key, while entry 3 stores the index of the LRU key. For extendable bucket tables, this field represents the next pointer (i.e. the pointer to the next group of 4 keys linked to the current bucket). The next pointer is not NULL if the bucket is currently extended or NULL otherwise.
3	Key [0 .. 3]	4 x key_size	Full keys.
4	Data [0 .. 3]	4 x entry_size	Full key values (key data) associated with keys 0 .. 3.

and detail the bucket search pipeline used to implement 8-byte and 16-byte key hash tables (either LRU or extendable bucket, either with pre-computed signature or "do-sig"). For each pipeline stage, the described operations are applied to each of the two packets handled by that stage.

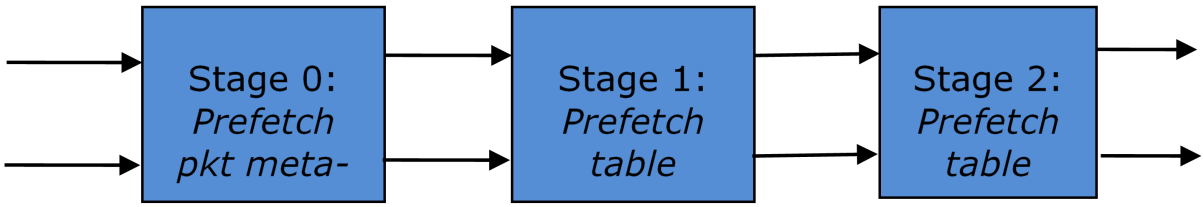


Fig. 5.85: Bucket Search Pipeline for Key Lookup Operation (Single Key Size Hash Tables)

Table 5.128: Description of the Bucket Search Pipeline Stages (8-byte and 16-byte Key Hash Tables)

#	Stage name	Description
0	Prefetch packet meta-data	1. Select next two packets from the burst of input packets. 2. Prefetch packet meta-data containing the key and key signature.
1	Prefetch table bucket	1. Read the key signature from the packet meta-data (for extendable bucket hash tables) or read the key from the packet meta-data and compute key signature (for LRU tables). 2. Identify the bucket ID using the key signature. 3. Prefetch the bucket.
2	Prefetch table data	1. Read the bucket. 2. Compare all 4 bucket keys against the input key. 3. Report input key as lookup hit only when a match is identified (more than one key match is not possible) 4. For LRU tables only, use branchless logic to update the bucket LRU list (the current key becomes the new MRU) only on lookup hit. 5. Prefetch the key value (key data) associated with the matched key (to avoid branches, this is done on both lookup hit and miss).

Additional notes:

1. The pipelined version of the bucket search algorithm is executed only if there are at least 5 packets in the burst of input packets. If there are less than 5 packets in the burst of input packets, a non-optimized implementation of the bucket search algorithm is executed.

2. For extendable bucket hash tables only, once the pipelined version of the bucket search algorithm has been executed for all the packets in the burst of input packets, the non-optimized implementation of the bucket search algorithm is also executed for any packets that did not produce a lookup hit, but have the bucket in extended state. As result of executing the non-optimized version, some of these packets may produce a lookup hit or lookup miss. This does not impact the performance of the key lookup operation, as the probability of having the bucket in extended state is relatively small.

5.46.5 Pipeline Library Design

A pipeline is defined by:

1. The set of input ports;
2. The set of output ports;
3. The set of tables;
4. The set of actions.

The input ports are connected with the output ports through tree-like topologies of interconnected tables. The table entries contain the actions defining the operations to be executed on the input packets and the packet flow within the pipeline.

Connectivity of Ports and Tables

To avoid any dependencies on the order in which pipeline elements are created, the connectivity of pipeline elements is defined after all the pipeline input ports, output ports and tables have been created.

General connectivity rules:

1. Each input port is connected to a single table. No input port should be left unconnected;
2. The table connectivity to other tables or to output ports is regulated by the next hop actions of each table entry and the default table entry. The table connectivity is fluid, as the table entries and the default table entry can be updated during run-time.
    - A table can have multiple entries (including the default entry) connected to the same output port. A table can have different entries connected to different output ports. Different tables can have entries (including default table entry) connected to the same output port.
    - A table can have multiple entries (including the default entry) connected to another table, in which case all these entries have to point to the same table. This constraint is enforced by the API and prevents tree-like topologies from being created (allowing table chaining only), with the purpose of simplifying the implementation of the pipeline run-time execution engine.

Port Actions

Port Action Handler

An action handler can be assigned to each input/output port to define actions to be executed on each input packet that is received by the port. Defining the action handler for a specific input/output port is optional (i.e. the action handler can be disabled).

For input ports, the action handler is executed after RX function. For output ports, the action handler is executed before the TX function.

The action handler can decide to drop packets.

Table Actions

Table Action Handler

An action handler to be executed on each input packet can be assigned to each table. Defining the action handler for a specific table is optional (i.e. the action handler can be disabled).

The action handler is executed after the table lookup operation is performed and the table entry associated with each input packet is identified. The action handler can only handle the user-defined actions, while the reserved actions (e.g. the next hop actions) are handled by the Packet Framework. The action handler can decide to drop the input packet.

Reserved Actions

The reserved actions are handled directly by the Packet Framework without the user being able to change their meaning through the table action handler configuration. A special category of the reserved actions is represented by the next hop actions, which regulate the packet flow between input ports, tables and output ports through the pipeline. Table 5.129 lists the next hop actions.

Table 5.129: Next Hop Actions (Reserved)

#	Next hop action	Description
1	Drop	Drop the current packet.
2	Send to output port	Send the current packet to specified output port. The output port ID is metadata stored in the same table entry.
3	Send to table	Send the current packet to specified table. The table ID is metadata stored in the same table entry.

User Actions

For each table, the meaning of user actions is defined through the configuration of the table action handler. Different tables can be configured with different action handlers, therefore the meaning of the user actions and their associated meta-data is private to each table. Within the same table, all the table entries (including the table default entry) share the same definition for the user actions and their associated meta-data, with each table entry having its own set of enabled user actions and its own copy of the action meta-data. Table 5.130 contains a non-exhaustive list of user action examples.

Table 5.130: User Action Examples

#	User action	Description
1	Metering	Per flow traffic metering using the srTCM and trTCM algorithms.
2	Statistics	Update the statistics counters maintained per flow.
3	App ID	Per flow state machine fed by variable length sequence of packets at the flow initialization with the purpose of identifying the traffic type and application.
4	Push/pop labels	Push/pop VLAN/MPLS labels to/from the current packet.
5	Network Address Translation (NAT)	Translate between the internal (LAN) and external (WAN) IP destination/source address and/or L4 protocol destination/source port.
6	TTL update	Decrement IP TTL and, in case of IPv4 packets, update the IP checksum.
7	Sym Crypto	Generate Cryptodev session based on the user-specified algorithm and key(s), and assemble the cryptodev operation based on the predefined offsets.

5.46.6 Multicore Scaling

A complex application is typically split across multiple cores, with cores communicating through SW queues. There is usually a performance limit on the number of table lookups and actions that can be fitted on the same CPU core due to HW constraints like: available CPU cycles, cache memory size, cache transfer BW, memory transfer BW, etc.

As the application is split across multiple CPU cores, the Packet Framework facilitates the creation of several pipelines, the assignment of each such pipeline to a different CPU core and the interconnection of all CPU core-level pipelines into a single application-level complex pipeline. For example, if CPU core A is assigned to run pipeline P1 and CPU core B pipeline P2, then the interconnection of P1 with P2 could be achieved by having the same set of SW queues act like output ports for P1 and input ports for P2.

This approach enables the application development using the pipeline, run-to-completion (clustered) or hybrid (mixed) models.

It is allowed for the same core to run several pipelines, but it is not allowed for several cores to run the same pipeline.

Shared Data Structures

The threads performing table lookup are actually table writers rather than just readers. Even if the specific table lookup algorithm is thread-safe for multiple readers (e. g. read-only access of the search algorithm data structures is enough to conduct the lookup operation), once the table entry for the current packet is identified, the thread is typically expected to update the action meta-data stored in the table entry (e.g. increment the counter tracking the number of packets that hit this table entry), and thus modify the table entry. During the time this thread is accessing this table entry (either writing or reading; duration is application specific), for data consistency reasons, no other threads (threads performing table lookup or entry add/delete operations) are allowed to modify this table entry.

Mechanisms to share the same table between multiple threads:

1. **Multiple writer threads.** Threads need to use synchronization primitives like semaphores (distinct semaphore per table entry) or atomic instructions. The cost of semaphores is usually high, even when the semaphore is free. The cost of atomic instructions is normally higher than the cost of regular instructions.
2. **Multiple writer threads, with single thread performing table lookup operations and multiple threads performing table entry add/delete operations.** The threads performing table entry add/delete operations send table update requests to the reader (typically through message passing queues), which does the actual table updates and then sends the response back to the request initiator.
3. **Single writer thread performing table entry add/delete operations and multiple reader threads that perform table lookup operations with read-only access to the table entries.** The reader threads use the main table copy while the writer is updating the mirror copy. Once the writer update is done, the writer can signal to the readers and busy wait until all readers swaps between the mirror copy (which now becomes the main copy) and the mirror copy (which now becomes the main copy).

5.46.7 Interfacing with Accelerators

The presence of accelerators is usually detected during the initialization phase by inspecting the HW devices that are part of the system (e.g. by PCI bus enumeration). Typical devices with acceleration capabilities are:

- Inline accelerators: NICs, switches, FPGAs, etc;
- Look-aside accelerators: chipsets, FPGAs, Intel QuickAssist, etc.

Usually, to support a specific functional block, specific implementation of Packet Framework tables and/or ports and/or actions has to be provided for each accelerator, with all the implementations sharing the same API: pure SW implementation (no acceleration), implementation using accelerator A, implementation using accelerator B, etc. The selection between these implementations could be done at build time or at run-time (recommended), based on which accelerators are present in the system, with no application changes required.

5.47 Vhost Library

The vhost library implements a user space virtio net server allowing the user to manipulate the virtio ring directly. In another words, it allows the user to fetch/put packets from/to the VM virtio net device. To achieve this, a vhost library should be able to:

- Access the guest memory:

  For QEMU, this is done by using the `-object memory-backend-file,share=on,...` option. Which means QEMU will create a file to serve as the guest RAM. The `share=on` option allows another process to map that file, which means it can access the guest RAM.

- Know all the necessary information about the vring:

  Information such as where the available ring is stored. Vhost defines some messages (passed through a Unix domain socket file) to tell the backend all the information it needs to know how to manipulate the vring.

5.47.1 Vhost API Overview

The following is an overview of some key Vhost API functions:

- `rte_vhost_driver_register(path, flags)`

  This function registers a vhost driver into the system. `path` specifies the Unix domain socket file path.

  Currently supported flags are:

  - `RTE_VHOST_USER_CLIENT`

    DPDK vhost-user will act as the client when this flag is given. See below for an explanation.

  - `RTE_VHOST_USER_NO_RECONNECT`

    When DPDK vhost-user acts as the client it will keep trying to reconnect to the server (QEMU) until it succeeds. This is useful in two cases:

    * When QEMU is not started yet.
    * When QEMU restarts (for example due to a guest OS reboot).

    This reconnect option is enabled by default. However, it can be turned off by setting this flag.

  - `RTE_VHOST_USER_DEQUEUE_ZERO_COPY`

    Dequeue zero copy will be enabled when this flag is set. It is disabled by default.

    There are some truths (including limitations) you might want to know while setting this flag:

    * zero copy is not good for small packets (typically for packet size below 512).
    * zero copy is really good for VM2VM case. For iperf between two VMs, the boost could be above 70% (when TSO is enabled).
    * For zero copy in VM2NIC case, guest Tx used vring may be starved if the PMD driver consume the mbuf but not release them timely.

      For example, i40e driver has an optimization to maximum NIC pipeline which postpones returning transmitted mbuf until only tx_free_threshold free descs left. The virtio

TX used ring will be starved if the formula (num_i40e_tx_desc - num_virtio_tx_desc > tx_free_threshold) is true, since i40e will not return back mbuf.

A performance tip for tuning zero copy in VM2NIC case is to adjust the frequency of mbuf free (i.e. adjust tx_free_threshold of i40e driver) to balance consumer and producer.

* Guest memory should be backended with huge pages to achieve better performance. Using 1G page size is the best.

  When dequeue zero copy is enabled, the guest phys address and host phys address mapping has to be established. Using non-huge pages means far more page segments. To make it simple, DPDK vhost does a linear search of those segments, thus the fewer the segments, the quicker we will get the mapping. NOTE: we may speed it by using tree searching in future.

* zero copy can not work when using vfio-pci with iommu mode currently, this is because we don't setup iommu dma mapping for guest memory. If you have to use vfio-pci driver, please insert vfio-pci kernel module in noiommu mode.

* The consumer of zero copy mbufs should consume these mbufs as soon as possible, otherwise it may block the operations in vhost.

- `RTE_VHOST_USER_IOMMU_SUPPORT`

  IOMMU support will be enabled when this flag is set. It is disabled by default.

  Enabling this flag makes possible to use guest vIOMMU to protect vhost from accessing memory the virtio device isn't allowed to, when the feature is negotiated and an IOMMU device is declared.

  However, this feature enables vhost-user's reply-ack protocol feature, which implementation is buggy in Qemu v2.7.0-v2.9.0 when doing multiqueue. Enabling this flag with these Qemu version results in Qemu being blocked when multiple queue pairs are declared.

- `RTE_VHOST_USER_POSTCOPY_SUPPORT`

  Postcopy live-migration support will be enabled when this flag is set. It is disabled by default.

  Enabling this flag should only be done when the calling application does not pre-fault the guest shared memory, otherwise migration would fail.

- `RTE_VHOST_USER_LINEARBUF_SUPPORT`

  Enabling this flag forces vhost dequeue function to only provide linear pktmbuf (no multi-segmented pktmbuf).

  The vhost library by default provides a single pktmbuf for given a packet, but if for some reason the data doesn't fit into a single pktmbuf (e.g., TSO is enabled), the library will allocate additional pktmbufs from the same mempool and chain them together to create a multi-segmented pktmbuf.

  However, the vhost application needs to support multi-segmented format. If the vhost application does not support that format and requires large buffers to be dequeue, this flag should be enabled to force only linear buffers (see RTE_VHOST_USER_EXTBUF_SUPPORT) or drop the packet.

  It is disabled by default.

- `RTE_VHOST_USER_EXTBUF_SUPPORT`

Enabling this flag allows vhost dequeue function to allocate and attach an external buffer to a pktmbuf if the pkmbuf doesn't provide enough space to store all data.

This is useful when the vhost application wants to support large packets but doesn't want to increase the default mempool object size nor to support multi-segmented mbufs (non-linear). In this case, a fresh buffer is allocated using rte_malloc() which gets attached to a pktmbuf using rte_pktmbuf_attach_extbuf().

See RTE_VHOST_USER_LINEARBUF_SUPPORT as well to disable multi-segmented mbufs for application that doesn't support chained mbufs.

It is disabled by default.

- `rte_vhost_driver_set_features(path, features)`

  This function sets the feature bits the vhost-user driver supports. The vhost-user driver could be vhost-user net, yet it could be something else, say, vhost-user SCSI.

- `rte_vhost_driver_callback_register(path, vhost_device_ops)`

  This function registers a set of callbacks, to let DPDK applications take the appropriate action when some events happen. The following events are currently supported:

  - `new_device(int vid)`

    This callback is invoked when a virtio device becomes ready. `vid` is the vhost device ID.

  - `destroy_device(int vid)`

    This callback is invoked when a virtio device is paused or shut down.

  - `vring_state_changed(int vid, uint16_t queue_id, int enable)`

    This callback is invoked when a specific queue's state is changed, for example to enabled or disabled.

  - `features_changed(int vid, uint64_t features)`

    This callback is invoked when the features is changed. For example, `VHOST_F_LOG_ALL` will be set/cleared at the start/end of live migration, respectively.

  - `new_connection(int vid)`

    This callback is invoked on new vhost-user socket connection. If DPDK acts as the server the device should not be deleted before `destroy_connection` callback is received.

  - `destroy_connection(int vid)`

    This callback is invoked when vhost-user socket connection is closed. It indicates that device with id `vid` is no longer in use and can be safely deleted.

- `rte_vhost_driver_disable/enable_features(path, features))`

  This function disables/enables some features. For example, it can be used to disable mergeable buffers and TSO features, which both are enabled by default.

- `rte_vhost_driver_start(path)`

  This function triggers the vhost-user negotiation. It should be invoked at the end of initializing a vhost-user driver.

- `rte_vhost_enqueue_burst(vid, queue_id, pkts, count)`

  Transmits (enqueues) `count` packets from host to guest.

- `rte_vhost_dequeue_burst(vid, queue_id, mbuf_pool, pkts, count)`

  Receives (dequeues) `count` packets from guest, and stored them at `pkts`.

- `rte_vhost_crypto_create(vid, cryptodev_id, sess_mempool, socket_id)`

  As an extension of new_device(), this function adds virtio-crypto workload acceleration capability to the device. All crypto workload is processed by DPDK cryptodev with the device ID of `cryptodev_id`.

- `rte_vhost_crypto_free(vid)`

  Frees the memory and vhost-user message handlers created in rte_vhost_crypto_create().

- `rte_vhost_crypto_fetch_requests(vid, queue_id, ops, nb_ops)`

  Receives (dequeues) `nb_ops` virtio-crypto requests from guest, parses them to DPDK Crypto Operations, and fills the `ops` with parsing results.

- `rte_vhost_crypto_finalize_requests(queue_id, ops, nb_ops)`

  After the `ops` are dequeued from Cryptodev, finalizes the jobs and notifies the guest(s).

- `rte_vhost_crypto_set_zero_copy(vid, option)`

  Enable or disable zero copy feature of the vhost crypto backend.

5.47.2 Vhost-user Implementations

Vhost-user uses Unix domain sockets for passing messages. This means the DPDK vhost-user implementation has two options:

- DPDK vhost-user acts as the server.

  DPDK will create a Unix domain socket server file and listen for connections from the frontend.

  Note, this is the default mode, and the only mode before DPDK v16.07.

- DPDK vhost-user acts as the client.

  Unlike the server mode, this mode doesn't create the socket file; it just tries to connect to the server (which responses to create the file instead).

  When the DPDK vhost-user application restarts, DPDK vhost-user will try to connect to the server again. This is how the "reconnect" feature works.

  Note:

  – The "reconnect" feature requires **QEMU v2.7** (or above).

  – The vhost supported features must be exactly the same before and after the restart. For example, if TSO is disabled and then enabled, nothing will work and issues undefined might happen.

No matter which mode is used, once a connection is established, DPDK vhost-user will start receiving and processing vhost messages from QEMU.

For messages with a file descriptor, the file descriptor can be used directly in the vhost process as it is already installed by the Unix domain socket.

The supported vhost messages are:

- VHOST_SET_MEM_TABLE
- VHOST_SET_VRING_KICK
- VHOST_SET_VRING_CALL
- VHOST_SET_LOG_FD
- VHOST_SET_VRING_ERR

For VHOST_SET_MEM_TABLE message, QEMU will send information for each memory region and its file descriptor in the ancillary data of the message. The file descriptor is used to map that region.

VHOST_SET_VRING_KICK is used as the signal to put the vhost device into the data plane, and VHOST_GET_VRING_BASE is used as the signal to remove the vhost device from the data plane.

When the socket connection is closed, vhost will destroy the device.

5.47.3 Guest memory requirement

- Memory pre-allocation

  For non-zerocopy, guest memory pre-allocation is not a must. This can help save of memory. If users really want the guest memory to be pre-allocated (e.g., for performance reason), we can add option -mem-prealloc when starting QEMU. Or, we can lock all memory at vhost side which will force memory to be allocated when mmap at vhost side; option –mlockall in ovs-dpdk is an example in hand.

  For zerocopy, we force the VM memory to be pre-allocated at vhost lib when mapping the guest memory; and also we need to lock the memory to prevent pages being swapped out to disk.

- Memory sharing

  Make sure share=on QEMU option is given. vhost-user will not work with a QEMU version without shared memory mapping.

5.47.4 Vhost supported vSwitch reference

For more vhost details and how to support vhost in vSwitch, please refer to the vhost example in the DPDK Sample Applications Guide.

5.47.5 Vhost data path acceleration (vDPA)

vDPA supports selective datapath in vhost-user lib by enabling virtio ring compatible devices to serve virtio driver directly for datapath acceleration.

rte_vhost_driver_attach_vdpa_device is used to configure the vhost device with accelerated backend.

Also vhost device capabilities are made configurable to adopt various devices. Such capabilities include supported features, protocol features, queue number.

Finally, a set of device ops is defined for device specific operations:

- `get_queue_num`

  Called to get supported queue number of the device.

- `get_features`

  Called to get supported features of the device.

- `get_protocol_features`

  Called to get supported protocol features of the device.

- `dev_conf`

  Called to configure the actual device when the virtio device becomes ready.

- `dev_close`

  Called to close the actual device when the virtio device is stopped.

- `set_vring_state`

  Called to change the state of the vring in the actual device when vring state changes.

- `set_features`

  Called to set the negotiated features to device.

- `migration_done`

  Called to allow the device to response to RARP sending.

- `get_vfio_group_fd`

  Called to get the VFIO group fd of the device.

- `get_vfio_device_fd`

  Called to get the VFIO device fd of the device.

- `get_notify_area`

  Called to get the notify area info of the queue.

5.48 Metrics Library

The Metrics library implements a mechanism by which *producers* can publish numeric information for later querying by *consumers*. In practice producers will typically be other libraries or primary processes, whereas consumers will typically be applications.

Metrics themselves are statistics that are not generated by PMDs. Metric information is populated using a push model, where producers update the values contained within the metric library by calling an update function on the relevant metrics. Consumers receive metric information by querying the central metric data, which is held in shared memory.

For each metric, a separate value is maintained for each port id, and when publishing metric values the producers need to specify which port is being updated. In addition there is a special id `RTE_METRICS_GLOBAL` that is intended for global statistics that are not associated with any individual device. Since the metrics library is self-contained, the only restriction on port numbers is that they are less than `RTE_MAX_ETHPORTS` - there is no requirement for the ports to actually exist.

5.48.1 Initializing the library

Before the library can be used, it has to be initialized by calling `rte_metrics_init()` which sets up the metric store in shared memory. This is where producers will publish metric information to, and where consumers will query it from.

```
rte_metrics_init(rte_socket_id());
```

This function **must** be called from a primary process, but otherwise producers and consumers can be in either primary or secondary processes.

5.48.2 Registering metrics

Metrics must first be *registered*, which is the way producers declare the names of the metrics they will be publishing. Registration can either be done individually, or a set of metrics can be registered as a group. Individual registration is done using `rte_metrics_reg_name()`:

```
id_1 = rte_metrics_reg_name("mean_bits_in");
id_2 = rte_metrics_reg_name("mean_bits_out");
id_3 = rte_metrics_reg_name("peak_bits_in");
id_4 = rte_metrics_reg_name("peak_bits_out");
```

or alternatively, a set of metrics can be registered together using `rte_metrics_reg_names()`:

```
const char * const names[] = {
    "mean_bits_in", "mean_bits_out",
    "peak_bits_in", "peak_bits_out",
};
id_set = rte_metrics_reg_names(&names[0], 4);
```

If the return value is negative, it means registration failed. Otherwise the return value is the *key* for the metric, which is used when updating values. A table mapping together these key values and the metrics' names can be obtained using `rte_metrics_get_names()`.

5.48.3 Updating metric values

Once registered, producers can update the metric for a given port using the `rte_metrics_update_value()` function. This uses the metric key that is returned when registering the metric, and can also be looked up using `rte_metrics_get_names()`.

```
rte_metrics_update_value(port_id, id_1, values[0]);
rte_metrics_update_value(port_id, id_2, values[1]);
rte_metrics_update_value(port_id, id_3, values[2]);
rte_metrics_update_value(port_id, id_4, values[3]);
```

if metrics were registered as a single set, they can either be updated individually using `rte_metrics_update_value()`, or updated together using the `rte_metrics_update_values()` function:

```
rte_metrics_update_value(port_id, id_set, values[0]);
rte_metrics_update_value(port_id, id_set + 1, values[1]);
rte_metrics_update_value(port_id, id_set + 2, values[2]);
rte_metrics_update_value(port_id, id_set + 3, values[3]);

rte_metrics_update_values(port_id, id_set, values, 4);
```

Note that `rte_metrics_update_values()` cannot be used to update metric values from *multiple sets*, as there is no guarantee two sets registered one after the other have contiguous id values.

5.48.4 Querying metrics

Consumers can obtain metric values by querying the metrics library using the `rte_metrics_get_values()` function that return an array of `struct rte_metric_value`. Each entry within this array contains a metric value and its associated key. A key-name mapping can be obtained using the `rte_metrics_get_names()` function that returns an array of `struct rte_metric_name` that is indexed by the key. The following will print out all metrics for a given port:

```
void print_metrics() {
    struct rte_metric_value *metrics;
    struct rte_metric_name *names;
    int len;

    len = rte_metrics_get_names(NULL, 0);
    if (len < 0) {
        printf("Cannot get metrics count\n");
        return;
    }
    if (len == 0) {
        printf("No metrics to display (none have been registered)\n");
        return;
    }
    metrics = malloc(sizeof(struct rte_metric_value) * len);
    names =  malloc(sizeof(struct rte_metric_name) * len);
    if (metrics == NULL || names == NULL) {
        printf("Cannot allocate memory\n");
        free(metrics);
        free(names);
        return;
    }
    ret = rte_metrics_get_values(port_id, metrics, len);
    if (ret < 0 || ret > len) {
        printf("Cannot get metrics values\n");
        free(metrics);
        free(names);
        return;
    }
    printf("Metrics for port %i:\n", port_id);
    for (i = 0; i < len; i++)
        printf("  %s: %"PRIu64"\n",
            names[metrics[i].key].name, metrics[i].value);
    free(metrics);
    free(names);
}
```

5.48.5 Deinitialising the library

Once the library usage is done, it must be deinitialized by calling `rte_metrics_deinit()` which will free the shared memory reserved during initialization.

```
err = rte_metrics_deinit(void);
```

If the return value is negative, it means deinitialization failed. This function **must** be called from a primary process.

5.48.6 Bit-rate statistics library

The bit-rate library calculates the exponentially-weighted moving average and peak bit-rates for each active port (i.e. network device). These statistics are reported via the metrics library using the following names:

- `mean_bits_in`: Average inbound bit-rate
- `mean_bits_out`: Average outbound bit-rate
- `ewma_bits_in`: Average inbound bit-rate (EWMA smoothed)
- `ewma_bits_out`: Average outbound bit-rate (EWMA smoothed)
- `peak_bits_in`: Peak inbound bit-rate
- `peak_bits_out`: Peak outbound bit-rate

Once initialised and clocked at the appropriate frequency, these statistics can be obtained by querying the metrics library.

Initialization

Before the library can be used, it has to be initialised by calling `rte_stats_bitrate_create()`, which will return a bit-rate calculation object. Since the bit-rate library uses the metrics library to report the calculated statistics, the bit-rate library then needs to register the calculated statistics with the metrics library. This is done using the helper function `rte_stats_bitrate_reg()`.

```
struct rte_stats_bitrates *bitrate_data;

bitrate_data = rte_stats_bitrate_create();
if (bitrate_data == NULL)
    rte_exit(EXIT_FAILURE, "Could not allocate bit-rate data.\n");
rte_stats_bitrate_reg(bitrate_data);
```

Controlling the sampling rate

Since the library works by periodic sampling but does not use an internal thread, the application has to periodically call `rte_stats_bitrate_calc()`. The frequency at which this function is called should be the intended sampling rate required for the calculated statistics. For instance if per-second statistics are desired, this function should be called once a second.

```
tics_datum = rte_rdtsc();
tics_per_1sec = rte_get_timer_hz();

while( 1 ) {
    /* ... */
    tics_current = rte_rdtsc();
    if (tics_current - tics_datum >= tics_per_1sec) {
        /* Periodic bitrate calculation */
        for (idx_port = 0; idx_port < cnt_ports; idx_port++)
                rte_stats_bitrate_calc(bitrate_data, idx_port);
            tics_datum = tics_current;
        }
    /* ... */
}
```

5.48.7 Latency statistics library

The latency statistics library calculates the latency of packet processing by a DPDK application, reporting the minimum, average, and maximum nano-seconds that packet processing takes, as well as the jitter in processing delay. These statistics are then reported via the metrics library using the following names:

- `min_latency_ns`: Minimum processing latency (nano-seconds)
- `avg_latency_ns`: Average processing latency (nano-seconds)
- `mac_latency_ns`: Maximum processing latency (nano-seconds)
- `jitter_ns`: Variance in processing latency (nano-seconds)

Once initialised and clocked at the appropriate frequency, these statistics can be obtained by querying the metrics library.

Initialization

Before the library can be used, it has to be initialised by calling `rte_latencystats_init()`.

```
lcoreid_t latencystats_lcore_id = -1;

int ret = rte_latencystats_init(1, NULL);
if (ret)
    rte_exit(EXIT_FAILURE, "Could not allocate latency data.\n");
```

Triggering statistic updates

The `rte_latencystats_update()` function needs to be called periodically so that latency statistics can be updated.

```
if (latencystats_lcore_id == rte_lcore_id())
    rte_latencystats_update();
```

Library shutdown

When finished, `rte_latencystats_uninit()` needs to be called to de-initialise the latency library.

```
rte_latencystats_uninit();
```

Timestamp and latency calculation

The Latency stats library marks the time in the timestamp field of the mbuf for the ingress packets and sets the `PKT_RX_TIMESTAMP` flag of `ol_flags` for the mbuf to indicate the marked time as a valid one. At the egress, the mbufs with the flag set are considered having valid timestamp and are used for the latency calculation.

5.49 Telemetry Library

The Telemetry library provides an interface to retrieve information from a variety of DPDK libraries. The library provides this information via socket connection, taking requests from a connected client and replying with the JSON response containing the requested telemetry information.

Telemetry is enabled to run by default when running a DPDK application, and the telemetry information from enabled libraries is made available. Libraries are responsible for registering their own commands, and providing the callback function that will format the library specific stats into the correct data format, when requested.

5.49.1 Registering Commands

Libraries and applications must register commands to make their information available via the Telemetry library. This involves providing a string command in the required format ("/library/command"), the callback function that will handle formatting the information when required, and help text for the command. An example showing ethdev commands being registered is shown below:

```
rte_telemetry_register_cmd("/ethdev/list", handle_port_list,
        "Returns list of available ethdev ports. Takes no parameters");
rte_telemetry_register_cmd("/ethdev/xstats", handle_port_xstats,
        "Returns the extended stats for a port. Parameters: int port_id");
rte_telemetry_register_cmd("/ethdev/link_status", handle_port_link_status,
        "Returns the link status for a port. Parameters: int port_id");
```

5.49.2 Formatting JSON response

The callback function provided by the library must format its telemetry information in the required data format. The Telemetry library provides a data utilities API to build up the response. For example, the ethdev library provides a list of available ethdev ports in a formatted data response, constructed using the following functions to build up the list:

```
rte_tel_data_start_array(d, RTE_TEL_INT_VAL);
    RTE_ETH_FOREACH_DEV(port_id)
        rte_tel_data_add_array_int(d, port_id);
```

The data structure is then formatted into a JSON response before sending. The resulting response shows the port list data provided above by the handler function in ethdev, placed in a JSON reply by telemetry:

```
{"/ethdev/list": [0, 1]}
```

For more information on the range of data functions available in the API, please refer to the docs.

5.50 Berkeley Packet Filter Library

The DPDK provides an BPF library that gives the ability to load and execute Enhanced Berkeley Packet Filter (eBPF) bytecode within user-space dpdk application.

It supports basic set of features from eBPF spec. Please refer to the *eBPF spec <https://www.kernel.org/doc/Documentation/networking/filter.txt>* for more information. Also it introduces basic framework to load/unload BPF-based filters on eth devices (right now only via SW RX/TX callbacks).

The library API provides the following basic operations:

- Create a new BPF execution context and load user provided eBPF code into it.
- Destroy an BPF execution context and its runtime structures and free the associated memory.
- Execute eBPF bytecode associated with provided input parameter.
- Provide information about natively compiled code for given BPF context.
- Load BPF program from the ELF file and install callback to execute it on given ethdev port/queue.

5.50.1 Not currently supported eBPF features

- JIT support only available for X86_64 and arm64 platforms
- cBPF
- tail-pointer call
- eBPF MAP
- skb
- external function calls for 32-bit platforms

5.51 IPsec Packet Processing Library

DPDK provides a library for IPsec data-path processing. The library utilizes the existing DPDK crypto-dev and security API to provide the application with a transparent and high performant IPsec packet processing API. The library is concentrated on data-path protocols processing (ESP and AH), IKE protocol(s) implementation is out of scope for this library.

5.51.1 SA level API

This API operates on the IPsec Security Association (SA) level. It provides functionality that allows user for given SA to process inbound and outbound IPsec packets.

To be more specific:

- for inbound ESP/AH packets perform decryption, authentication, integrity checking, remove ESP/AH related headers
- for outbound packets perform payload encryption, attach ICV, update/add IP headers, add ESP/AH headers/trailers,
- setup related mbuf fields (ol_flags, tx_offloads, etc.).
- initialize/un-initialize given SA based on user provided parameters.

The SA level API is based on top of crypto-dev/security API and relies on them to perform actual cipher and integrity checking.

Due to the nature of the crypto-dev API (enqueue/dequeue model) the library introduces an asynchronous API for IPsec packets destined to be processed by the crypto-device.

The expected API call sequence for data-path processing would be:

```
/* enqueue for processing by crypto-device */
rte_ipsec_pkt_crypto_prepare(...);
rte_cryptodev_enqueue_burst(...);
/* dequeue from crypto-device and do final processing (if any) */
rte_cryptodev_dequeue_burst(...);
rte_ipsec_pkt_crypto_group(...); /* optional */
rte_ipsec_pkt_process(...);
```

For packets destined for inline processing no extra overhead is required and the synchronous API call: rte_ipsec_pkt_process() is sufficient for that case.

Note: For more details about the IPsec API, please refer to the *DPDK API Reference*.

The current implementation supports all four currently defined rte_security types:

RTE_SECURITY_ACTION_TYPE_NONE

In that mode the library functions perform

- for inbound packets:
  - check SQN
  - prepare *rte_crypto_op* structure for each input packet
  - verify that integrity check and decryption performed by crypto device completed successfully
  - check padding data
  - remove outer IP header (tunnel mode) / update IP header (transport mode)
  - remove ESP header and trailer, padding, IV and ICV data
  - update SA replay window

- for outbound packets:
  - generate SQN and IV
  - add outer IP header (tunnel mode) / update IP header (transport mode)
  - add ESP header and trailer, padding and IV data
  - prepare *rte_crypto_op* structure for each input packet
  - verify that crypto device operations (encryption, ICV generation) were completed successfully

RTE_SECURITY_ACTION_TYPE_CPU_CRYPTO

In that mode the library functions perform same operations as in `RTE_SECURITY_ACTION_TYPE_NONE`. The only difference is that crypto operations are performed with CPU crypto synchronous API.

RTE_SECURITY_ACTION_TYPE_INLINE_CRYPTO

In that mode the library functions perform

- for inbound packets:
  - verify that integrity check and decryption performed by *rte_security* device completed successfully
  - check SQN
  - check padding data
  - remove outer IP header (tunnel mode) / update IP header (transport mode)
  - remove ESP header and trailer, padding, IV and ICV data
  - update SA replay window
- for outbound packets:
  - generate SQN and IV
  - add outer IP header (tunnel mode) / update IP header (transport mode)
  - add ESP header and trailer, padding and IV data
  - update *ol_flags* inside *struct rte_mbuf* to indicate that inline-crypto processing has to be performed by HW on this packet
  - invoke *rte_security* device specific *set_pkt_metadata()* to associate security device specific data with the packet

RTE_SECURITY_ACTION_TYPE_INLINE_PROTOCOL

In that mode the library functions perform

- for inbound packets:
  - verify that integrity check and decryption performed by *rte_security* device completed successfully
- for outbound packets:
  - update *ol_flags* inside *struct rte_mbuf* to indicate that inline-crypto processing has to be performed by HW on this packet
  - invoke *rte_security* device specific *set_pkt_metadata()* to associate security device specific data with the packet

RTE_SECURITY_ACTION_TYPE_LOOKASIDE_PROTOCOL

In that mode the library functions perform

- for inbound packets:
  - prepare *rte_crypto_op* structure for each input packet
  - verify that integrity check and decryption performed by crypto device completed successfully
- for outbound packets:
  - prepare *rte_crypto_op* structure for each input packet
  - verify that crypto device operations (encryption, ICV generation) were completed successfully

To accommodate future custom implementations function pointers model is used for both *crypto_prepare* and *process* implementations.

5.51.2 SA database API

SA database(SAD) is a table with <key, value> pairs.

Value is an opaque user provided pointer to the user defined SA data structure.

According to RFC4301 each SA can be uniquely identified by a key which is either:

- security parameter index(SPI)
- or SPI and destination IP(DIP)
- or SPI, DIP and source IP(SIP)

In case of multiple matches, longest matching key will be returned.

Create/destroy

librte_ipsec SAD implementation provides ability to create/destroy SAD tables.

To create SAD table user has to specify how many entries of each key type is required and IP protocol type (IPv4/IPv6). As an example:

```
struct rte_ipsec_sad *sad;
struct rte_ipsec_sad_conf conf;

conf.socket_id = -1;
conf.max_sa[RTE_IPSEC_SAD_SPI_ONLY] = some_nb_rules_spi_only;
conf.max_sa[RTE_IPSEC_SAD_SPI_DIP] = some_nb_rules_spi_dip;
conf.max_sa[RTE_IPSEC_SAD_SPI_DIP_SIP] = some_nb_rules_spi_dip_sip;
conf.flags = RTE_IPSEC_SAD_FLAG_RW_CONCURRENCY;

sad = rte_ipsec_sad_create("test", &conf);
```

Note: for more information please refer to ipsec library API reference

Add/delete rules

Library also provides methods to add or delete key/value pairs from the SAD. To add user has to specify key, key type and a value which is an opaque pointer to SA. The key type reflects a set of tuple fields that will be used for lookup of the SA. As mentioned above there are 3 types of a key and the representation of a key type is:

```
RTE_IPSEC_SAD_SPI_ONLY,
RTE_IPSEC_SAD_SPI_DIP,
RTE_IPSEC_SAD_SPI_DIP_SIP,
```

As an example, to add new entry into the SAD for IPv4 addresses:

```
struct rte_ipsec_sa *sa;
union rte_ipsec_sad_key key;

key.v4.spi = rte_cpu_to_be_32(spi_val);
if (key_type >= RTE_IPSEC_SAD_SPI_DIP) /* DIP is optional*/
    key.v4.dip = rte_cpu_to_be_32(dip_val);
if (key_type == RTE_IPSEC_SAD_SPI_DIP_SIP) /* SIP is optional*/
    key.v4.sip = rte_cpu_to_be_32(sip_val);

rte_ipsec_sad_add(sad, &key, key_type, sa);
```

Note: By performance reason it is better to keep spi/dip/sip in net byte order to eliminate byteswap on lookup

To delete user has to specify key and key type.

Delete code would look like:

```
union rte_ipsec_sad_key key;
```

(continues on next page)

(continued from previous page)

```
key.v4.spi = rte_cpu_to_be_32(necessary_spi);
if (key_type >= RTE_IPSEC_SAD_SPI_DIP) /* DIP is optional*/
    key.v4.dip = rte_cpu_to_be_32(necessary_dip);
if (key_type == RTE_IPSEC_SAD_SPI_DIP_SIP) /* SIP is optional*/
    key.v4.sip = rte_cpu_to_be_32(necessary_sip);

rte_ipsec_sad_del(sad, &key, key_type);
```

Lookup

Library provides lookup by the given {SPI,DIP,SIP} tuple of inbound ipsec packet as a key.

The search key is represented by:

```
union rte_ipsec_sad_key {
    struct rte_ipsec_sadv4_key  v4;
    struct rte_ipsec_sadv6_key  v6;
};
```

where v4 is a tuple for IPv4:

```
struct rte_ipsec_sadv4_key {
    uint32_t spi;
    uint32_t dip;
    uint32_t sip;
};
```

and v6 is a tuple for IPv6:

```
struct rte_ipsec_sadv6_key {
    uint32_t spi;
    uint8_t dip[16];
    uint8_t sip[16];
};
```

As an example, lookup related code could look like that:

```
int i;
union rte_ipsec_sad_key keys[BURST_SZ];
const union rte_ipsec_sad_key *keys_p[BURST_SZ];
void *vals[BURST_SZ];

for (i = 0; i < BURST_SZ_MAX; i++) {
    keys[i].v4.spi = esp_hdr[i]->spi;
    keys[i].v4.dip = ipv4_hdr[i]->dst_addr;
    keys[i].v4.sip = ipv4_hdr[i]->src_addr;
    keys_p[i] = &keys[i];
}
rte_ipsec_sad_lookup(sad, keys_p, vals, BURST_SZ);

for (i = 0; i < BURST_SZ_MAX; i++) {
    if (vals[i] == NULL)
        printf("SA not found for key index %d\n", i);
    else
        printf("SA pointer is %p\n", vals[i]);
}
```

5.51.3 Supported features

- ESP protocol tunnel mode both IPv4/IPv6.
- ESP protocol transport mode both IPv4/IPv6.
- ESN and replay window.
- algorithms: 3DES-CBC, AES-CBC, AES-CTR, AES-GCM, HMAC-SHA1, NULL.

5.51.4 Limitations

The following features are not properly supported in the current version:

- Hard/soft limit for SA lifetime (time interval/byte count).

5.52 Graph Library and Inbuilt Nodes

Graph architecture abstracts the data processing functions as a `node` and `links` them together to create a complex `graph` to enable reusable/modular data processing functions.

The graph library provides API to enable graph framework operations such as create, lookup, dump and destroy on graph and node operations such as clone, edge update, and edge shrink, etc. The API also allows to create the stats cluster to monitor per graph and per node stats.

5.52.1 Features

Features of the Graph library are:

- Nodes as plugins.
- Support for out of tree nodes.
- Inbuilt nodes for packet processing.
- Multi-process support.
- Low overhead graph walk and node enqueue.
- Low overhead statistics collection infrastructure.
- Support to export the graph as a Graphviz dot file. See `rte_graph_export()`.
- Allow having another graph walk implementation in the future by segregating the fast path(`rte_graph_worker.h`) and slow path code.

5.52.2 Advantages of Graph architecture

- Memory latency is the enemy for high-speed packet processing, moving the similar packet processing code to a node will reduce the I cache and D caches misses.
- Exploits the probability that most packets will follow the same nodes in the graph.
- Allow SIMD instructions for packet processing of the node.-
- The modular scheme allows having reusable nodes for the consumers.
- The modular scheme allows us to abstract the vendor HW specific optimizations as a node.

5.52.3 Performance tuning parameters

- Test with various burst size values (256, 128, 64, 32) using CONFIG_RTE_GRAPH_BURST_SIZE config option. The testing shows, on x86 and arm64 servers, The sweet spot is 256 burst size. While on arm64 embedded SoCs, it is either 64 or 128.
- Disable node statistics (using `CONFIG_RTE_LIBRTE_GRAPH_STATS` config option) if not needed.
- Use arm64 optimized memory copy for arm64 architecture by selecting `CONFIG_RTE_ARCH_ARM64_MEMCPY`.

5.52.4 Programming model

Anatomy of Node:

The `figure_anatomy_of_a_node` diagram depicts the anatomy of a node.

The node is the basic building block of the graph framework.

A node consists of:

process():

The callback function will be invoked by worker thread using `rte_graph_walk()` function when there is data to be processed by the node. A graph node process the function using `process()` and enqueue to next downstream node using `rte_node_enqueue*()` function.

Context memory:

It is memory allocated by the library to store the node-specific context information. This memory will be used by process(), init(), fini() callbacks.

init():

The callback function will be invoked by `rte_graph_create()` on when a node gets attached to a graph.

fini():

The callback function will be invoked by `rte_graph_destroy()` on when a node gets detached to a graph.

Node name:

It is the name of the node. When a node registers to graph library, the library gives the ID as `rte_node_t` type. Both ID or Name shall be used lookup the node. `rte_node_from_name()`, `rte_node_id_to_name()` are the node lookup functions.

nb_edges:

The number of downstream nodes connected to this node. The `next_nodes[]` stores the downstream nodes objects. `rte_node_edge_update()` and `rte_node_edge_shrink()` functions shall be used to update the `next_node[]` objects. Consumers of the node APIs are free to update the `next_node[]` objects till `rte_graph_create()` invoked.

next_node[]:

The dynamic array to store the downstream nodes connected to this node. Downstream node should not be current node itself or a source node.

Source node:

Source nodes are static nodes created using `RTE_NODE_REGISTER` by passing `flags` as `RTE_NODE_SOURCE_F`. While performing the graph walk, the `process()` function of all the source nodes will be called first. So that these nodes can be used as input nodes for a graph.

Node creation and registration

- Node implementer creates the node by implementing ops and attributes of `struct rte_node_register`.
- The library registers the node by invoking RTE_NODE_REGISTER on library load using the constructor scheme. The constructor scheme used here to support multi-process.

Link the Nodes to create the graph topology

The figure_link_the_nodes diagram shows a graph topology after linking the N nodes.

Once nodes are available to the program, Application or node public API functions can links them together to create a complex packet processing graph.

There are multiple different types of strategies to link the nodes.

Method (a):

Provide the next_nodes[] at the node registration time. See struct rte_node_register::nb_edges. This is a use case to address the static node scheme where one knows upfront the next_nodes[] of the node.

Method (b):

Use rte_node_edge_get(), rte_node_edge_update(), rte_node_edge_shrink() to update the next_nodes[] links for the node runtime but before graph create.

Method (c):

Use rte_node_clone() to clone a already existing node, created using RTE_NODE_REGISTER. When rte_node_clone() invoked, The library, would clone all the attributes of the node and creates a new one. The name for cloned node shall be "parent_node_name-user_provided_name".

This method enables the use case of Rx and Tx nodes where multiple of those nodes need to be cloned based on the number of CPU available in the system. The cloned nodes will be identical, except the "context memory". Context memory will have information of port, queue pair in case of Rx and Tx ethdev nodes.

Create the graph object

Now that the nodes are linked, Its time to create a graph by including the required nodes. The application can provide a set of node patterns to form a graph object. The famish() API used underneath for the pattern matching to include the required nodes. After the graph create any changes to nodes or graph is not allowed.

The rte_graph_create() API shall be used to create the graph.

Example of a graph object creation:

```
{"ethdev_rx-0-0", ip4*, ethdev_tx-*"}
```

In the above example, A graph object will be created with ethdev Rx node of port 0 and queue 0, all ipv4* nodes in the system, and ethdev tx node of all ports.

Multicore graph processing

In the current graph library implementation, specifically, `rte_graph_walk()` and `rte_node_enqueue*` fast path API functions are designed to work on single-core to have better performance. The fast path API works on graph object, So the multi-core graph processing strategy would be to create graph object PER WORKER.

In fast path

Typical fast-path code looks like below, where the application gets the fast-path graph object using `rte_graph_lookup()` on the worker thread and run the `rte_graph_walk()` in a tight loop.

```
struct rte_graph *graph = rte_graph_lookup("worker0");

while (!done) {
    rte_graph_walk(graph);
}
```

Context update when graph walk in action

The fast-path object for the node is `struct rte_node`.

It may be possible that in slow-path or after the graph walk-in action, the user needs to update the context of the node hence access to `struct rte_node *` memory.

`rte_graph_foreach_node()`, `rte_graph_node_get()`, `rte_graph_node_get_by_name()` APIs can be used to to get the `struct rte_node*`. `rte_graph_foreach_node()` iterator function works on `struct rte_graph *` fast-path graph object while others works on graph ID or name.

Get the node statistics using graph cluster

The user may need to know the aggregate stats of the node across multiple graph objects. Especially the situation where each graph object bound to a worker thread.

Introduced a graph cluster object for statistics. `rte_graph_cluster_stats_create()` API shall be used for creating a graph cluster with multiple graph objects and `rte_graph_cluster_stats_get()` to get the aggregate node statistics.

An example statistics output from `rte_graph_cluster_stats_get()`

```
+---------+-----------+-------------+---------------+-----------+---------------+-----------+
|Node     |calls      |objs         |realloc_count  |objs/call  |objs/sec(10E6) |cycles/call|
+---------+-----------+-------------+---------------+-----------+---------------+-----------+
|node0    |12977424   |3322220544   |5              |256.000    |3047.151872    |20.0000    |
|node1    |12977653   |3322279168   |0              |256.000    |3047.210496    |17.0000    |
|node2    |12977696   |3322290176   |0              |256.000    |3047.221504    |17.0000    |
|node3    |12977734   |3322299904   |0              |256.000    |3047.231232    |17.0000    |
|node4    |12977784   |3322312704   |1              |256.000    |3047.243776    |17.0000    |
|node5    |12977825   |3322323200   |0              |256.000    |3047.254528    |17.0000    |
+---------+-----------+-------------+---------------+-----------+---------------+-----------+
```

Node writing guidelines

The `process()` function of a node is the fast-path function and that needs to be written carefully to achieve max performance.

Broadly speaking, there are two different types of nodes.

Static nodes

The first kind of nodes are those that have a fixed `next_nodes[]` for the complete burst (like ethdev_rx, ethdev_tx) and it is simple to write. `process()` function can move the obj burst to the next node either using `rte_node_next_stream_move()` or using `rte_node_next_stream_get()` and `rte_node_next_stream_put()`.

Intermediate nodes

The second kind of such node is `intermediate nodes` that decide what is the `next_node[]` to send to on a per-packet basis. In these nodes,

- Firstly, there has to be the best possible packet processing logic.
- Secondly, each packet needs to be queued to its next node.

This can be done using `rte_node_enqueue_[x1|x2|x4]()` APIs if they are to single next or `rte_node_enqueue_next()` that takes array of nexts.

In scenario where multiple intermediate nodes are present but most of the time each node using the same next node for all its packets, the cost of moving every pointer from current node's stream to next node's stream could be avoided. This is called home run and `rte_node_next_stream_move()` could be used to just move stream from the current node to the next node with least number of cycles. Since this can be avoided only in the case where all the packets are destined to the same next node, node implementation should be also having worst-case handling where every packet could be going to different next node.

Example of intermediate node implementation with home run:

1. Start with speculation that next_node = node->ctx. This could be the next_node application used in the previous function call of this node.

2. Get the next_node stream array with required space using `rte_node_next_stream_get(next_node, space)`.

3. while n_left_from > 0 (i.e packets left to be sent) prefetch next pkt_set and process current pkt_set to find their next node

4. if all the next nodes of the current pkt_set match speculated next node, just count them as successfully speculated(`last_spec`) till now and continue the loop without actually moving them to the next node. else if there is a mismatch, copy all the pkt_set pointers that were `last_spec` and move the current pkt_set to their respective next's nodes using `rte_enqueue_next_x1()`. Also, one of the next_node can be updated as speculated next_node if it is more probable. Finally, reset `last_spec` to zero.

5. if n_left_from != 0 then goto 3) to process remaining packets.

6. if last_spec == nb_objs, All the objects passed were successfully speculated to single next node. So, the current stream can be moved to next node using `rte_node_next_stream_move(node, next_node)`. This is the `home run` where memcpy of buffer pointers to next node is avoided.

7. Update the `node->ctx` with more probable next node.

5.52.5 Graph object memory layout

The `figure_graph_mem_layout` diagram shows `rte_graph` object memory layout. Understanding the memory layout helps to debug the graph library and improve the performance if needed.

Graph object consists of a header, circular buffer to store the pending stream when walking over the graph, and variable-length memory to store the `rte_node` objects.

The graph_nodes_mem_create() creates and populate this memory. The functions such as `rte_graph_walk()` and `rte_node_enqueue_*` use this memory to enable fastpath services.

5.52.6 Inbuilt Nodes

DPDK provides a set of nodes for data processing. The following section details the documentation for the same.

ethdev_rx

This node does `rte_eth_rx_burst()` into stream buffer passed to it (src node stream) and does `rte_node_next_stream_move()` only when there are packets received. Each `rte_node` works only on one Rx port and queue that it gets from node->ctx. For each (port X, rx_queue Y), a rte_node is cloned from ethdev_rx_base_node as `ethdev_rx-X-Y` in `rte_node_eth_config()` along with updating `node->ctx`. Each graph needs to be associated with a unique rte_node for a (port, rx_queue).

ethdev_tx

This node does `rte_eth_tx_burst()` for a burst of objs received by it. It sends the burst to a fixed Tx Port and Queue information from node->ctx. For each (port X), this `rte_node` is cloned from ethdev_tx_node_base as "ethdev_tx-X" in `rte_node_eth_config()` along with updating node->context.

Since each graph doesn't need more than one Txq, per port, a Txq is assigned based on graph id to each rte_node instance. Each graph needs to be associated with a rte_node for each (port).

pkt_drop

This node frees all the objects passed to it considering them as `rte_mbufs` that need to be freed.

ip4_lookup

This node is an intermediate node that does LPM lookup for the received ipv4 packets and the result determines each packets next node.

On successful LPM lookup, the result contains the `next_node` id and `next-hop` id with which the packet needs to be further processed.

On LPM lookup failure, objects are redirected to pkt_drop node. `rte_node_ip4_route_add()` is control path API to add ipv4 routes. To achieve home run, node use `rte_node_stream_move()` as mentioned in above sections.

ip4_rewrite

This node gets packets from `ip4_lookup` node with next-hop id for each packet is embedded in `node_mbuf_priv1(mbuf)->nh`. This id is used to determine the L2 header to be written to the packet before sending the packet out to a particular ethdev_tx node. `rte_node_ip4_rewrite_add()` is control path API to add next-hop info.

null

This node ignores the set of objects passed to it and reports that all are processed.

Part 2: Development Environment

5.53 Source Organization

This section describes the organization of sources in the DPDK framework.

5.53.1 Makefiles and Config

Note: In the following descriptions, `RTE_SDK` is the environment variable that points to the base directory into which the tarball was extracted. See *Useful Variables Provided by the Build System* for descriptions of other variables.

Makefiles that are provided by the DPDK libraries and applications are located in `$(RTE_SDK)/mk`.

Config templates are located in `$(RTE_SDK)/config`. The templates describe the options that are enabled for each target. The config file also contains items that can be enabled and disabled for many of the DPDK libraries, including debug options. The user should look at the config file and become familiar with these options. The config file is also used to create a header file, which will be located in the new build directory.

5.53.2 Libraries

Libraries are located in subdirectories of `$(RTE_SDK)/lib`. By convention a library refers to any code that provides an API to an application. Typically, it generates an archive file (`.a`), but a kernel module would also go in the same directory.

5.53.3 Drivers

Drivers are special libraries which provide poll-mode driver implementations for devices: either hardware devices or pseudo/virtual devices. They are contained in the *drivers* subdirectory, classified by type, and each compiles to a library with the format `librte_pmd_X.a` where `X` is the driver name.

Note: Several of the `driver/net` directories contain a `base` sub-directory. The `base` directory generally contains code the shouldn't be modified directly by the user. Any enhancements should be done via the `X_osdep.c` and/or `X_osdep.h` files in that directory. Refer to the local README in the base directories for driver specific instructions.

5.53.4 Applications

Applications are source files that contain a `main()` function. They are located in the `$(RTE_SDK)/app` and `$(RTE_SDK)/examples` directories.

The app directory contains sample applications that are used to test DPDK (such as autotests) or the Poll Mode Drivers (test-pmd).

The examples directory contains *Sample applications* that show how libraries can be used.

5.54 Development Kit Build System

The DPDK requires a build system for compilation activities and so on. This section describes the constraints and the mechanisms used in the DPDK framework.

There are two use-cases for the framework:

- Compilation of the DPDK libraries and sample applications; the framework generates specific binary libraries, include files and sample applications
- Compilation of an external application or library, using an installed binary DPDK

5.54.1 Building the Development Kit Binary

The following provides details on how to build the DPDK binary.

Build Directory Concept

After installation, a build directory structure is created. Each build directory contains include files, libraries, and applications.

A build directory is specific to a configuration that includes architecture + execution environment + toolchain. It is possible to have several build directories sharing the same sources with different configurations.

For instance, to create a new build directory called my_sdk_build_dir using the default configuration template config/defconfig_x86_64-linux, we use:

```
cd ${RTE_SDK}
make config T=x86_64-native-linux-gcc O=my_sdk_build_dir
```

This creates a new my_sdk_build_dir directory. After that, we can compile by doing:

```
cd my_sdk_build_dir
make
```

which is equivalent to:

```
make O=my_sdk_build_dir
```

Refer to *Development Kit Root Makefile Help* for details about make commands that can be used from the root of DPDK.

5.54.2 Building External Applications

Since DPDK is in essence a development kit, the first objective of end users will be to create an application using this SDK. To compile an application, the user must set the RTE_SDK and RTE_TARGET environment variables.

```
export RTE_SDK=/opt/DPDK
export RTE_TARGET=x86_64-native-linux-gcc
cd /path/to/my_app
```

For a new application, the user must create their own Makefile that includes some .mk files, such as ${RTE_SDK}/mk/rte.vars.mk, and ${RTE_SDK}/mk/ rte.app.mk. This is described in *Building Your Own Application*.

Depending on the chosen target (architecture, machine, executive environment, toolchain) defined in the Makefile or as an environment variable, the applications and libraries will compile using the appropriate .h files and will link with the appropriate .a files. These files are located in ${RTE_SDK}/arch-machine-execenv-toolchain, which is referenced internally by ${RTE_BIN_SDK}.

To compile their application, the user just has to call make. The compilation result will be located in /path/to/my_app/build directory.

Sample applications are provided in the examples directory.

5.54.3 Makefile Description

General Rules For DPDK Makefiles

In the DPDK, Makefiles always follow the same scheme:

1. Include $(RTE_SDK)/mk/rte.vars.mk at the beginning.
2. Define specific variables for RTE build system.
3. Include a specific $(RTE_SDK)/mk/rte.XYZ.mk, where XYZ can be app, lib, extapp, extlib, obj, gnuconfigure, and so on, depending on what kind of object you want to build. *See Makefile Types* below.
4. Include user-defined rules and variables.

   The following is a very simple example of an external application Makefile:

```
include $(RTE_SDK)/mk/rte.vars.mk

# binary name
APP = helloworld

# all source are stored in SRCS-y
SRCS-y := main.c

CFLAGS += -O3
CFLAGS += $(WERROR_FLAGS)

include $(RTE_SDK)/mk/rte.extapp.mk
```

Makefile Types

Depending on the .mk file which is included at the end of the user Makefile, the Makefile will have a different role. Note that it is not possible to build a library and an application in the same Makefile. For that, the user must create two separate Makefiles, possibly in two different directories.

In any case, the rte.vars.mk file must be included in the user Makefile as soon as possible.

Application

These Makefiles generate a binary application.

- rte.app.mk: Application in the development kit framework
- rte.extapp.mk: External application
- rte.hostapp.mk: prerequisite tool to build dpdk

Library

Generate a .a library.

- rte.lib.mk: Library in the development kit framework
- rte.extlib.mk: external library
- rte.hostlib.mk: host library in the development kit framework

Install

- rte.install.mk: Does not build anything, it is only used to create links or copy files to the installation directory. This is useful for including files in the development kit framework.

Kernel Module

- rte.module.mk: Build a kernel module in the development kit framework.

Objects

- rte.obj.mk: Object aggregation (merge several .o in one) in the development kit framework.
- rte.extobj.mk: Object aggregation (merge several .o in one) outside the development kit framework.

Misc

- rte.gnuconfigure.mk: Build an application that is configure-based.
- rte.subdir.mk: Build several directories in the development kit framework.

Internally Generated Build Tools

`app/dpdk-pmdinfogen`

`dpdk-pmdinfogen` scans an object (.o) file for various well known symbol names. These well known symbol names are defined by various macros and used to export important information about hardware support and usage for pmd files. For instance the macro:

```
RTE_PMD_REGISTER_PCI(name, drv)
```

Creates the following symbol:

```
static char this_pmd_name0[] __attribute__((used)) = "<name>";
```

Which `dpdk-pmdinfogen` scans for. Using this information other relevant bits of data can be exported from the object file and used to produce a hardware support description, that `dpdk-pmdinfogen` then encodes into a JSON formatted string in the following format:

```
static char <name_pmd_string>="PMD_INFO_STRING=\"{'name' : '<name>', ...}\"";
```

These strings can then be searched for by external tools to determine the hardware support of a given library or application.

Useful Variables Provided by the Build System

- RTE_SDK: The absolute path to the DPDK sources. When compiling the development kit, this variable is automatically set by the framework. It has to be defined by the user as an environment variable if compiling an external application.
- RTE_SRCDIR: The path to the root of the sources. When compiling the development kit, RTE_SRCDIR = RTE_SDK. When compiling an external application, the variable points to the root of external application sources.
- RTE_OUTPUT: The path to which output files are written. Typically, it is $(RTE_SRCDIR)/build, but it can be overridden by the O= option in the make command line.
- RTE_TARGET: A string identifying the target for which we are building. The format is arch-machine-execenv-toolchain. When compiling the SDK, the target is deduced by the build system from the configuration (.config). When building an external application, it must be specified by the user in the Makefile or as an environment variable.
- RTE_SDK_BIN: References $(RTE_SDK)/$(RTE_TARGET).
- RTE_ARCH: Defines the architecture (i686, x86_64). It is the same value as CONFIG_RTE_ARCH but without the double-quotes around the string.
- RTE_MACHINE: Defines the machine. It is the same value as CONFIG_RTE_MACHINE but without the double-quotes around the string.
- RTE_TOOLCHAIN: Defines the toolchain (gcc , icc). It is the same value as CONFIG_RTE_TOOLCHAIN but without the double-quotes around the string.
- RTE_EXEC_ENV: Defines the executive environment (linux). It is the same value as CONFIG_RTE_EXEC_ENV but without the double-quotes around the string.
- RTE_KERNELDIR: This variable contains the absolute path to the kernel sources that will be used to compile the kernel modules. The kernel headers must be the same as the ones that will be used on the target machine (the machine that will run the application). By default, the variable is set to /lib/modules/$(shell uname -r)/build, which is correct when the target machine is also the build machine.
- RTE_DEVEL_BUILD: Stricter options (stop on warning). It defaults to y in a git tree.

Variables that Can be Set/Overridden in a Makefile Only

- VPATH: The path list that the build system will search for sources. By default, RTE_SRCDIR will be included in VPATH.
- CFLAGS: Flags to use for C compilation. The user should use += to append data in this variable.
- LDFLAGS: Flags to use for linking. The user should use += to append data in this variable.
- ASFLAGS: Flags to use for assembly. The user should use += to append data in this variable.
- CPPFLAGS: Flags to use to give flags to C preprocessor (only useful when assembling .S files). The user should use += to append data in this variable.

- LDLIBS: In an application, the list of libraries to link with (for example, -L /path/to/libfoo -lfoo). The user should use += to append data in this variable.
- SRC-y: A list of source files (.c, .S, or .o if the source is a binary) in case of application, library or object Makefiles. The sources must be available from VPATH.
- INSTALL-y-$(INSTPATH): A list of files to be installed in $(INSTPATH). The files must be available from VPATH and will be copied in $(RTE_OUTPUT)/$(INSTPATH). Can be used in almost any RTE Makefile.
- SYMLINK-y-$(INSTPATH): A list of files to be installed in $(INSTPATH). The files must be available from VPATH and will be linked (symbolically) in $(RTE_OUTPUT)/$(INSTPATH). This variable can be used in almost any DPDK Makefile.
- PREBUILD: A list of prerequisite actions to be taken before building. The user should use += to append data in this variable.
- POSTBUILD: A list of actions to be taken after the main build. The user should use += to append data in this variable.
- PREINSTALL: A list of prerequisite actions to be taken before installing. The user should use += to append data in this variable.
- POSTINSTALL: A list of actions to be taken after installing. The user should use += to append data in this variable.
- PRECLEAN: A list of prerequisite actions to be taken before cleaning. The user should use += to append data in this variable.
- POSTCLEAN: A list of actions to be taken after cleaning. The user should use += to append data in this variable.
- DEPDIRS-$(DIR): Only used in the development kit framework to specify if the build of the current directory depends on build of another one. This is needed to support parallel builds correctly.

Variables that can be Set/Overridden by the User on the Command Line Only

Some variables can be used to configure the build system behavior. They are documented in *Development Kit Root Makefile Help* and *External Application/Library Makefile Help*

- WERROR_CFLAGS: By default, this is set to a specific value that depends on the compiler. Users are encouraged to use this variable as follows:

  CFLAGS += $(WERROR_CFLAGS)

This avoids the use of different cases depending on the compiler (icc or gcc). Also, this variable can be overridden from the command line, which allows bypassing of the flags for testing purposes.

Variables that Can be Set/Overridden by the User in a Makefile or Command Line

- CFLAGS_my_file.o: Specific flags to add for C compilation of my_file.c.
- LDFLAGS_my_app: Specific flags to add when linking my_app.
- EXTRA_CFLAGS: The content of this variable is appended after CFLAGS when compiling.
- EXTRA_LDFLAGS: The content of this variable is appended after LDFLAGS when linking.
- EXTRA_LDLIBS: The content of this variable is appended after LDLIBS when linking.
- EXTRA_ASFLAGS: The content of this variable is appended after ASFLAGS when assembling.
- EXTRA_CPPFLAGS: The content of this variable is appended after CPPFLAGS when using a C preprocessor on assembly files.

5.55 Development Kit Root Makefile Help

The DPDK provides a root level Makefile with targets for configuration, building, cleaning, testing, installation and others. These targets are explained in the following sections.

5.55.1 Configuration Targets

The configuration target requires the name of the target, which is specified using T=mytarget and it is mandatory. The list of available targets are in $(RTE_SDK)/config (remove the defconfig _ prefix).

Configuration targets also support the specification of the name of the output directory, using O=mybuilddir. This is an optional parameter, the default output directory is build.

- Config

  This will create a build directory, and generates a configuration from a template. A Makefile is also created in the new build directory.

  Example:

```
make config O=mybuild T=x86_64-native-linux-gcc
```

5.55.2 Build Targets

Build targets support the optional specification of the name of the output directory, using O=mybuilddir. The default output directory is build.

- all, build or just make

  Build the DPDK in the output directory previously created by a make config.

  Example:

```
make O=mybuild
```

- clean

  Clean all objects created using make build.

  Example:

```
make clean O=mybuild
```

- %_sub

  Build a subdirectory only, without managing dependencies on other directories.

  Example:

```
make lib/librte_eal_sub O=mybuild
```

- %_clean

  Clean a subdirectory only.

  Example:

```
make lib/librte_eal_clean O=mybuild
```

5.55.3 Install Targets

- Install

  The list of available targets are in $(RTE_SDK)/config (remove the defconfig_ prefix).

  The GNU standards variables may be used: http://gnu.org/prep/standards/html_node/Directory-Variables.html and http://gnu.org/prep/standards/html_node/DESTDIR.html

  Example:

```
make install DESTDIR=myinstall prefix=/usr
```

5.55.4 Test Targets

- test

  Launch automatic tests for a build directory specified using O=mybuilddir. It is optional, the default output directory is build.

  Example:

```
make test O=mybuild
```

5.55.5 Documentation Targets

- doc

  Generate the documentation (API and guides).

- doc-api-html

  Generate the Doxygen API documentation in html.

- doc-guides-html

  Generate the guides documentation in html.

- doc-guides-pdf

  Generate the guides documentation in pdf.

5.55.6 Misc Targets

- help

  Show a quick help.

5.55.7 Other Useful Command-line Variables

The following variables can be specified on the command line:

- V=

  Enable verbose build (show full compilation command line, and some intermediate commands).

- D=

  Enable dependency debugging. This provides some useful information about why a target is built or not.

- EXTRA_CFLAGS=, EXTRA_LDFLAGS=, EXTRA_LDLIBS=, EXTRA_ASFLAGS=, EXTRA_CPPFLAGS=

  Append specific compilation, link or asm flags.

- CROSS=

  Specify a cross toolchain header that will prefix all gcc/binutils applications. This only works when using gcc.

5.55.8 Make in a Build Directory

All targets described above are called from the SDK root $(RTE_SDK). It is possible to run the same Makefile targets inside the build directory. For instance, the following command:

```
cd $(RTE_SDK)
make config O=mybuild T=x86_64-native-linux-gcc
make O=mybuild
```

is equivalent to:

```
cd $(RTE_SDK)
make config O=mybuild T=x86_64-native-linux-gcc
cd mybuild

# no need to specify O= now
make
```

5.55.9 Compiling for Debug

To compile the DPDK and sample applications with debugging information included and the optimization level set to 0, the EXTRA_CFLAGS environment variable should be set before compiling as follows:

```
export EXTRA_CFLAGS='-O0 -g'
```

5.56 Installing DPDK Using the meson build system

5.56.1 Summary

For many platforms, compiling and installing DPDK should work using the following set of commands:

```
meson build
cd build
ninja
ninja install
```

This will compile DPDK in the `build` subdirectory, and then install the resulting libraries, drivers and header files onto the system - generally in /usr/local. A package-config file, `libdpdk.pc`, for DPDK will also be installed to allow ease of compiling and linking with applications.

After installation, to use DPDK, the necessary CFLAG and LDFLAG variables can be got from pkg-config:

```
pkg-config --cflags libdpdk
pkg-config --libs libdpdk
```

More detail on each of these steps can be got from the following sections.

5.56.2 Getting the Tools

The `meson` tool is used to configure a DPDK build. On most Linux distributions this can be got using the local package management system, e.g. `dnf install meson` or `apt-get install meson`. If meson is not available as a suitable package, it can also be installed using the Python 3 `pip` tool, e.g. `pip3 install meson`. Version 0.47.1 of meson is required - if the version packaged is too old, the latest version is generally available from "pip".

The other dependency for building is the `ninja` tool, which acts similar to make and performs the actual build using information provided by meson. Installing meson will, in many cases, also install ninja, but, if not already installed, it too is generally packaged by most Linux distributions. If not available as a package, it can be downloaded as source or binary from https://ninja-build.org/

5.56.3 Configuring the Build

To configure a build, run the meson tool, passing the path to the directory to be used for the build e.g. `meson build`, as shown above. If calling meson from somewhere other than the root directory of the DPDK project the path to the root directory should be passed as the first parameter, and the build path as the second. For example, to build DPDK in /tmp/dpdk-build:

```
user@host:/tmp$ meson ~user/dpdk dpdk-build
```

Meson will then configure the build based on settings in the project's meson.build files, and by checking the build environment for e.g. compiler properties or the presence of dependencies, such as libpcap, or openssl libcrypto libraries. Once done, meson writes a `build.ninja` file in the build directory to be used to do the build itself when ninja is called.

Tuning of the build is possible, both as part of the original meson call, or subsequently using `meson configure` command (`mesonconf` in some older versions). Some options, such as `buildtype`, or `werror` are built into meson, while others, such as `max_lcores`, or the list of examples to build, are DPDK-specific. To have a list of all options available run `meson configure` in the build directory.

Examples of adjusting the defaults when doing initial meson configuration. Project-specific options are passed used -Doption=value:

```
meson --werror werrorbuild  # build with warnings as errors

meson --buildtype=debug debugbuild  # build for debugging

meson -Dexamples=l3fwd,l2fwd fwdbuild  # build some examples as
                                # part of the normal DPDK build

meson -Dmax_lcores=8 smallbuild  # scale build for smaller systems

meson -Denable_docs=true fullbuild  # build and install docs

meson -Dmachine=default  # use builder-independent baseline -march

meson -Ddisable_drivers=event/*,net/tap  # disable tap driver and all
                                # eventdev PMDs for a smaller build

meson -Denable_trace_fp=true tracebuild # build with fast path traces
                                # enabled
```

Examples of setting some of the same options using meson configure:

```
meson configure -Dwerror=true

meson configure -Dbuildtype=debug

meson configure -Dexamples=l3fwd,l2fwd

meson configure -Dmax_lcores=8

meson configure -Denable_trace_fp=true
```

NOTE: once meson has been run to configure a build in a directory, it cannot be run again on the same directory. Instead `meson configure` should be used to change the build settings within the directory, and when `ninja` is called to do the build itself, it will trigger the necessary re-scan from meson.

NOTE: machine=default uses a config that works on all supported architectures regardless of the capabilities of the machine where the build is happening.

As well as those settings taken from `meson configure`, other options such as the compiler to use can be passed via environment variables. For example:

```
CC=clang meson clang-build
```

NOTE: for more comprehensive overriding of compilers or other environment settings, the tools for cross-compilation may be considered. However, for basic overriding of the compiler etc., the above form works as expected.

5.56.4 Performing the Build

Use `ninja` to perform the actual build inside the build folder previously configured. In most cases no arguments are necessary.

Ninja accepts a number of flags which are similar to make. For example, to call ninja from outside the build folder, you can use `ninja -C build`. Ninja also runs parallel builds by default, but you can limit this using the `-j` flag, e.g. `ninja -j1 -v` to do the build one step at a time, printing each command on a new line as it runs.

5.56.5 Installing the Compiled Files

Use `ninja install` to install the required DPDK files onto the system. The install prefix defaults to `/usr/local` but can be used as with other options above. The environment variable `DESTDIR` can be used to adjust the root directory for the install, for example when packaging.

With the base install directory, the individual directories for libraries and headers are configurable. By default, the following will be the installed layout:

```
headers -> /usr/local/include
libraries -> /usr/local/lib64
drivers -> /usr/local/lib64/dpdk/drivers
libdpdk.pc -> /usr/local/lib64/pkgconfig
```

For the drivers, these will also be symbolically linked into the library install directory, so that ld.so can find them in cases where one driver may depend on another, e.g. a NIC PMD depending upon the PCI bus driver. Within the EAL, the default search path for drivers will be set to the configured driver install path, so dynamically-linked applications can be run without having to pass in `-d /path/to/driver` options for standard drivers.

5.56.6 Cross Compiling DPDK

To cross-compile DPDK on a desired target machine we can use the following command:

```
meson cross-build --cross-file <target_machine_configuration>
```

For example if the target machine is arm64 we can use the following command:

```
meson arm-build --cross-file config/arm/arm64_armv8_linux_gcc
```

where config/arm/arm64_armv8_linux_gcc contains settings for the compilers and other build tools to be used, as well as characteristics of the target machine.

5.56.7 Using the DPDK within an Application

To compile and link against DPDK within an application, pkg-config should be used to query the correct parameters. Examples of this are given in the makefiles for the example applications included with DPDK. They demonstrate how to link either against the DPDK shared libraries, or against the static versions of the same.

From examples/helloworld/Makefile:

```
PC_FILE := $(shell pkg-config --path libdpdk)
CFLAGS += -O3 $(shell pkg-config --cflags libdpdk)
LDFLAGS_SHARED = $(shell pkg-config --libs libdpdk)
LDFLAGS_STATIC = -Wl,-Bstatic $(shell pkg-config --static --libs libdpdk)

build/$(APP)-shared: $(SRCS-y) Makefile $(PC_FILE) | build
        $(CC) $(CFLAGS) $(SRCS-y) -o $@ $(LDFLAGS) $(LDFLAGS_SHARED)

build/$(APP)-static: $(SRCS-y) Makefile $(PC_FILE) | build
        $(CC) $(CFLAGS) $(SRCS-y) -o $@ $(LDFLAGS) $(LDFLAGS_STATIC)

build:
        @mkdir -p $@
```

5.57 Running DPDK Unit Tests with Meson

This section describes how to run test cases with the DPDK meson build system.

Steps to build and install DPDK using meson can be referred in *Installing DPDK Using the meson build system*

5.57.1 Grouping of test cases

Test cases have been classified into four different groups.

- Fast tests.
- Performance tests.
- Driver tests.
- Tests which produce lists of objects as output, and therefore that need manual checking.

These tests can be run using the argument to `meson test` as `--suite project_name:label`.

For example:

```
$ meson test -C <build path> --suite DPDK:fast-tests
```

If the `<build path>` is current working directory, the `-C <build path>` option can be skipped as below:

```
$ meson test --suite DPDK:fast-tests
```

The project name is optional so the following is equivalent to the previous command:

```
$ meson test --suite fast-tests
```

The meson command to list all available tests:

```
$ meson test --list
```

Test cases are run serially by default for better stability.

Arguments of `test()` that can be provided in meson.build are as below:

- `is_parallel` is used to run test case either in parallel or non-parallel mode.
- `timeout` is used to specify the timeout of test case.
- `args` is used to specify test specific parameters.
- `env` is used to specify test specific environment parameters.

5.57.2 Dealing with skipped test cases

Some unit test cases have a dependency on external libraries, driver modules or config flags, without which the test cases cannot be run. Such test cases will be reported as skipped if they cannot run. To enable those test cases, the user should ensure the required dependencies are met. Below are a few possible causes why tests may be skipped:

1. Optional external libraries are not found.
2. Config flags for the dependent library are not enabled.
3. Dependent driver modules are not installed on the system.
4. Not enough processing cores. Some tests are skipped on machines with 2 or 4 cores.

5.58 Extending the DPDK

This chapter describes how a developer can extend the DPDK to provide a new library, a new target, or support a new target.

5.58.1 Example: Adding a New Library libfoo

To add a new library to the DPDK, proceed as follows:

1. Add a new configuration option:

```
for f in config/\*; do \
    echo CONFIG_RTE_LIBFOO=y >> $f; done
```

2. Create a new directory with sources:

```
mkdir ${RTE_SDK}/lib/libfoo
touch ${RTE_SDK}/lib/libfoo/foo.c
touch ${RTE_SDK}/lib/libfoo/foo.h
```

3. Add a foo() function in libfoo.

   Definition is in foo.c:

```
void foo(void)
{
}
```

Declaration is in foo.h:

```
extern void foo(void);
```

4. Update lib/Makefile:

```
vi ${RTE_SDK}/lib/Makefile
# add:
# DIRS-$(CONFIG_RTE_LIBFOO) += libfoo
```

5. Create a new Makefile for this library, for example, derived from mempool Makefile:

```
cp ${RTE_SDK}/lib/librte_mempool/Makefile ${RTE_SDK}/lib/libfoo/

vi ${RTE_SDK}/lib/libfoo/Makefile
# replace:
# librte_mempool -> libfoo
# rte_mempool -> foo
```

6. Update mk/DPDK.app.mk, and add -lfoo in LDLIBS variable when the option is enabled. This will automatically add this flag when linking a DPDK application.

7. Build the DPDK with the new library (we only show a specific target here):

```
cd ${RTE_SDK}
make config T=x86_64-native-linux-gcc
make
```

8. Check that the library is installed:

```
ls build/lib
ls build/include
```

Example: Using libfoo in the Test Application

The test application is used to validate all functionality of the DPDK. Once you have added a library, a new test case should be added in the test application.

- A new test_foo.c file should be added, that includes foo.h and calls the foo() function from test_foo(). When the test passes, the test_foo() function should return 0.
- Makefile, test.h and commands.c must be updated also, to handle the new test case.
- Test report generation: autotest.py is a script that is used to generate the test report that is available in the ${RTE_SDK}/doc/rst/test_report/autotests directory. This script must be updated also. If libfoo is in a new test family, the links in ${RTE_SDK}/doc/rst/test_report/test_report.rst must be updated.
- Build the DPDK with the updated test application (we only show a specific target here):

```
cd ${RTE_SDK}
make config T=x86_64-native-linux-gcc
make
```

5.59 Building Your Own Application

5.59.1 Compiling a Sample Application in the Development Kit Directory

When compiling a sample application (for example, hello world), the following variables must be exported: RTE_SDK and RTE_TARGET.

```
~/DPDK$ cd examples/helloworld/
~/DPDK/examples/helloworld$ export RTE_SDK=/home/user/DPDK
~/DPDK/examples/helloworld$ export RTE_TARGET=x86_64-native-linux-gcc
~/DPDK/examples/helloworld$ make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map
```

The binary is generated in the build directory by default:

```
~/DPDK/examples/helloworld$ ls build/app
helloworld helloworld.map
```

5.59.2 Build Your Own Application Outside the Development Kit

The sample application (Hello World) can be duplicated in a new directory as a starting point for your development:

```
~$ cp -r DPDK/examples/helloworld my_rte_app
~$ cd my_rte_app/
~/my_rte_app$ export RTE_SDK=/home/user/DPDK
~/my_rte_app$ export RTE_TARGET=x86_64-native-linux-gcc
~/my_rte_app$ make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map
```

5.59.3 Customizing Makefiles

Application Makefile

The default makefile provided with the Hello World sample application is a good starting point. It includes:

- $(RTE_SDK)/mk/rte.vars.mk at the beginning
- $(RTE_SDK)/mk/rte.extapp.mk at the end

The user must define several variables:

- APP: Contains the name of the application.
- SRCS-y: List of source files (*.c, *.S).

Library Makefile

It is also possible to build a library in the same way:

- Include $(RTE_SDK)/mk/rte.vars.mk at the beginning.
- Include $(RTE_SDK)/mk/rte.extlib.mk at the end.

The only difference is that APP should be replaced by LIB, which contains the name of the library. For example, libfoo.a.

Customize Makefile Actions

Some variables can be defined to customize Makefile actions. The most common are listed below. Refer to *Makefile Description* section in *Development Kit Build System*

chapter for details.

- VPATH: The path list where the build system will search for sources. By default, RTE_SRCDIR will be included in VPATH.
- CFLAGS_my_file.o: The specific flags to add for C compilation of my_file.c.
- CFLAGS: The flags to use for C compilation.
- LDFLAGS: The flags to use for linking.
- CPPFLAGS: The flags to use to provide flags to the C preprocessor (only useful when assembling .S files)
- LDLIBS: A list of libraries to link with (for example, -L /path/to/libfoo - lfoo)

5.60 External Application/Library Makefile help

External applications or libraries should include specific Makefiles from RTE_SDK, located in mk directory. These Makefiles are:

- ${RTE_SDK}/mk/rte.extapp.mk: Build an application
- ${RTE_SDK}/mk/rte.extlib.mk: Build a static library
- ${RTE_SDK}/mk/rte.extobj.mk: Build objects (.o)

5.60.1 Prerequisites

The following variables must be defined:

- ${RTE_SDK}: Points to the root directory of the DPDK.
- ${RTE_TARGET}: Reference the target to be used for compilation (for example, x86_64-native-linux-gcc).

5.60.2 Build Targets

Build targets support the specification of the name of the output directory, using O=mybuilddir. This is optional; the default output directory is build.

- all, "nothing" (meaning just make)

  Build the application or the library in the specified output directory.

  Example:

```
make O=mybuild
```

- clean

  Clean all objects created using make build.

  Example:

```
make clean O=mybuild
```

5.60.3 Help Targets

- help

  Show this help.

5.60.4 Other Useful Command-line Variables

The following variables can be specified at the command line:

- S=

  Specify the directory in which the sources are located. By default, it is the current directory.

- M=

  Specify the Makefile to call once the output directory is created. By default, it uses $(S)/Makefile.

- V=

  Enable verbose build (show full compilation command line and some intermediate commands).

- D=

  Enable dependency debugging. This provides some useful information about why a target must be rebuilt or not.

- EXTRA_CFLAGS=, EXTRA_LDFLAGS=, EXTRA_ASFLAGS=, EXTRA_CPPFLAGS=

  Append specific compilation, link or asm flags.

- CROSS=

  Specify a cross-toolchain header that will prefix all gcc/binutils applications. This only works when using gcc.

5.60.5 Make from Another Directory

It is possible to run the Makefile from another directory, by specifying the output and the source dir. For example:

```
export RTE_SDK=/path/to/DPDK
export RTE_TARGET=x86_64-native-linux-icc
make -f /path/to/my_app/Makefile S=/path/to/my_app O=/path/to/build_dir
```

Part 3: Performance Optimization

5.61 Performance Optimization Guidelines

5.61.1 Introduction

The following sections describe optimizations used in DPDK and optimizations that should be considered for new applications.

They also highlight the performance-impacting coding techniques that should, and should not be, used when developing an application using the DPDK.

And finally, they give an introduction to application profiling using a Performance Analyzer from Intel to optimize the software.

5.62 Writing Efficient Code

This chapter provides some tips for developing efficient code using the DPDK. For additional and more general information, please refer to the *Intel® 64 and IA-32 Architectures Optimization Reference Manual* which is a valuable reference to writing efficient code.

5.62.1 Memory

This section describes some key memory considerations when developing applications in the DPDK environment.

Memory Copy: Do not Use libc in the Data Plane

Many libc functions are available in the DPDK, via the Linux* application environment. This can ease the porting of applications and the development of the configuration plane. However, many of these functions are not designed for performance. Functions such as memcpy() or strcpy() should not be used in the data plane. To copy small structures, the preference is for a simpler technique that can be optimized by the compiler. Refer to the *VTune™ Performance Analyzer Essentials* publication from Intel Press for recommendations.

For specific functions that are called often, it is also a good idea to provide a self-made optimized function, which should be declared as static inline.

The DPDK API provides an optimized rte_memcpy() function.

Memory Allocation

Other functions of libc, such as malloc(), provide a flexible way to allocate and free memory. In some cases, using dynamic allocation is necessary, but it is really not advised to use malloc-like functions in the data plane because managing a fragmented heap can be costly and the allocator may not be optimized for parallel allocation.

If you really need dynamic allocation in the data plane, it is better to use a memory pool of fixed-size objects. This API is provided by librte_mempool. This data structure provides several services that increase performance, such as memory alignment of objects, lockless access to objects, NUMA awareness, bulk get/put and per-lcore cache. The rte_malloc () function uses a similar concept to mempools.

Concurrent Access to the Same Memory Area

Read-Write (RW) access operations by several lcores to the same memory area can generate a lot of data cache misses, which are very costly. It is often possible to use per-lcore variables, for example, in the case of statistics. There are at least two solutions for this:

- Use RTE_PER_LCORE variables. Note that in this case, data on lcore X is not available to lcore Y.
- Use a table of structures (one per lcore). In this case, each structure must be cache-aligned.

Read-mostly variables can be shared among lcores without performance losses if there are no RW variables in the same cache line.

NUMA

On a NUMA system, it is preferable to access local memory since remote memory access is slower. In the DPDK, the memzone, ring, rte_malloc and mempool APIs provide a way to create a pool on a specific socket.

Sometimes, it can be a good idea to duplicate data to optimize speed. For read-mostly variables that are often accessed, it should not be a problem to keep them in one socket only, since data will be present in cache.

Distribution Across Memory Channels

Modern memory controllers have several memory channels that can load or store data in parallel. Depending on the memory controller and its configuration, the number of channels and the way the memory is distributed across the channels varies. Each channel has a bandwidth limit, meaning that if all memory access operations are done on the first channel only, there is a potential bottleneck.

By default, the *Mempool Library* spreads the addresses of objects among memory channels.

Locking memory pages

The underlying operating system is allowed to load/unload memory pages at its own discretion. These page loads could impact the performance, as the process is on hold when the kernel fetches them.

To avoid these you could pre-load, and lock them into memory with the `mlockall()` call.

```
if (mlockall(MCL_CURRENT | MCL_FUTURE)) {
    RTE_LOG(NOTICE, USER1, "mlockall() failed with error \"%s\"\n",
            strerror(errno));
}
```

5.62.2 Communication Between lcores

To provide a message-based communication between lcores, it is advised to use the DPDK ring API, which provides a lockless ring implementation.

The ring supports bulk and burst access, meaning that it is possible to read several elements from the ring with only one costly atomic operation (see *Ring Library*). Performance is greatly improved when using bulk access operations.

The code algorithm that dequeues messages may be something similar to the following:

```
#define MAX_BULK 32

while (1) {
    /* Process as many elements as can be dequeued. */
    count = rte_ring_dequeue_burst(ring, obj_table, MAX_BULK, NULL);
    if (unlikely(count == 0))
        continue;

    my_process_bulk(obj_table, count);
}
```

5.62.3 PMD Driver

The DPDK Poll Mode Driver (PMD) is also able to work in bulk/burst mode, allowing the factorization of some code for each call in the send or receive function.

Avoid partial writes. When PCI devices write to system memory through DMA, it costs less if the write operation is on a full cache line as opposed to part of it. In the PMD code, actions have been taken to avoid partial writes as much as possible.

Lower Packet Latency

Traditionally, there is a trade-off between throughput and latency. An application can be tuned to achieve a high throughput, but the end-to-end latency of an average packet will typically increase as a result. Similarly, the application can be tuned to have, on average, a low end-to-end latency, at the cost of lower throughput.

In order to achieve higher throughput, the DPDK attempts to aggregate the cost of processing each packet individually by processing packets in bursts.

Using the testpmd application as an example, the burst size can be set on the command line to a value of 16 (also the default value). This allows the application to request 16 packets at a time from the PMD.

The testpmd application then immediately attempts to transmit all the packets that were received, in this case, all 16 packets.

The packets are not transmitted until the tail pointer is updated on the corresponding TX queue of the network port. This behavior is desirable when tuning for high throughput because the cost of tail pointer updates to both the RX and TX queues can be spread across 16 packets, effectively hiding the relatively slow MMIO cost of writing to the PCIe* device. However, this is not very desirable when tuning for low latency because the first packet that was received must also wait for another 15 packets to be received. It cannot be transmitted until the other 15 packets have also been processed because the NIC will not know to transmit the packets until the TX tail pointer has been updated, which is not done until all 16 packets have been processed for transmission.

To consistently achieve low latency, even under heavy system load, the application developer should avoid processing packets in bunches. The testpmd application can be configured from the command line to use a burst value of 1. This will allow a single packet to be processed at a time, providing lower latency, but with the added cost of lower throughput.

5.62.4 Locks and Atomic Operations

Atomic operations imply a lock prefix before the instruction, causing the processor's LOCK# signal to be asserted during execution of the following instruction. This has a big impact on performance in a multicore environment.

Performance can be improved by avoiding lock mechanisms in the data plane. It can often be replaced by other solutions like per-lcore variables. Also, some locking techniques are more efficient than others. For instance, the Read-Copy-Update (RCU) algorithm can frequently replace simple rwlocks.

5.62.5 Coding Considerations

Inline Functions

Small functions can be declared as static inline in the header file. This avoids the cost of a call instruction (and the associated context saving). However, this technique is not always efficient; it depends on many factors including the compiler.

Branch Prediction

The Intel® C/C++ Compiler (icc)/gcc built-in helper functions likely() and unlikely() allow the developer to indicate if a code branch is likely to be taken or not. For instance:

```
if (likely(x > 1))
    do_stuff();
```

5.62.6 Setting the Target CPU Type

The DPDK supports CPU microarchitecture-specific optimizations by means of CONFIG_RTE_MACHINE option in the DPDK configuration file. The degree of optimization depends on the compiler's ability to optimize for a specific microarchitecture, therefore it is preferable to use the latest compiler versions whenever possible.

If the compiler version does not support the specific feature set (for example, the Intel® AVX instruction set), the build process gracefully degrades to whatever latest feature set is supported by the compiler.

Since the build and runtime targets may not be the same, the resulting binary also contains a platform check that runs before the main() function and checks if the current machine is suitable for running the binary.

Along with compiler optimizations, a set of preprocessor defines are automatically added to the build process (regardless of the compiler version). These defines correspond to the instruction sets that the target CPU should be able to support. For example, a binary compiled for any SSE4.2-capable processor will have RTE_MACHINE_CPUFLAG_SSE4_2 defined, thus enabling compile-time code path selection for different platforms.

5.63 Link Time Optimization

The DPDK supports compilation with link time optimization turned on. This depends obviously on the ability of the compiler to do "whole program" optimization at link time and is available only for compilers that support that feature. To be more specific, compiler (in addition to performing LTO) have to support creation of ELF objects containing both normal code and internal representation (called fat-lto-objects in gcc and icc). This is required since during build some code is generated by parsing produced ELF objects (pmdinfogen).

The amount of performance gain that one can get from LTO depends on the compiler and the code that is being compiled. However LTO is also useful for additional code analysis done by the compiler. In particular due to interprocedural analysis compiler can produce additional warnings about variables that might be used uninitialized. Some of these warnings might be "false positives" though and you might need to explicitly initialize variable in order to silence the compiler.

Please note that turning LTO on causes considerable extension of build time.

When using make based build, link time optimization can be enabled for the whole DPDK by setting:

```
CONFIG_RTE_ENABLE_LTO=y
```

in config file.

For the meson based build it can be enabled by setting meson built-in 'b_lto' option:

```
meson build -Db_lto=true
```

5.64 Profile Your Application

The following sections describe methods of profiling DPDK applications on different architectures.

5.64.1 Profiling on x86

Intel processors provide performance counters to monitor events. Some tools provided by Intel, such as Intel® VTune™ Amplifier, can be used to profile and benchmark an application. See the *VTune Performance Analyzer Essentials* publication from Intel Press for more information.

For a DPDK application, this can be done in a Linux* application environment only.

The main situations that should be monitored through event counters are:

- Cache misses
- Branch mis-predicts
- DTLB misses
- Long latency instructions and exceptions

Refer to the Intel Performance Analysis Guide for details about application profiling.

Profiling with VTune

To allow VTune attaching to the DPDK application, reconfigure and recompile the DPDK with `CONFIG_RTE_ETHDEV_RXTX_CALLBACKS` and `CONFIG_RTE_ETHDEV_PROFILE_WITH_VTUNE` enabled.

5.64.2 Profiling on ARM64

Using Linux perf

The ARM64 architecture provide performance counters to monitor events. The Linux `perf` tool can be used to profile and benchmark an application. In addition to the standard events, `perf` can be used to profile arm64 specific PMU (Performance Monitor Unit) events through raw events (`-e -rXX`).

For more derails refer to the ARM64 specific PMU events enumeration.

Low-resolution generic counter

The default `cntvct_el0` based `rte_rdtsc()` provides a portable means to get a wall clock counter in user space. Typically it runs at a lower clock frequency than the CPU clock frequency. Cycles counted using this method should be scaled to CPU clock frequency.

High-resolution cycle counter

The alternative method to enable `rte_rdtsc()` for a high resolution wall clock counter is through the ARMv8 PMU subsystem. The PMU cycle counter runs at CPU frequency. However, access to the PMU cycle counter from user space is not enabled by default in the arm64 linux kernel. It is possible to enable cycle counter for user space access by configuring the PMU from the privileged mode (kernel space).

By default the `rte_rdtsc()` implementation uses a portable `cntvct_el0` scheme. Application can choose the PMU based implementation with `CONFIG_RTE_ARM_EAL_RDTSC_USE_PMU`.

The example below shows the steps to configure the PMU based cycle counter on an ARMv8 machine.

```
git clone https://github.com/jerinjacobk/armv8_pmu_cycle_counter_el0
cd armv8_pmu_cycle_counter_el0
make
sudo insmod pmu_el0_cycle_counter.ko
cd $DPDK_DIR
make config T=arm64-armv8a-linux-gcc
echo "CONFIG_RTE_ARM_EAL_RDTSC_USE_PMU=y" >> build/.config
make
```

Warning: The PMU based scheme is useful for high accuracy performance profiling with `rte_rdtsc()`. However, this method can not be used in conjunction with Linux userspace profiling tools like `perf` as this scheme alters the PMU registers state.

5.65 Glossary

ACL
Access Control List

API
Application Programming Interface

ASLR
Linux* kernel Address-Space Layout Randomization

BSD
Berkeley Software Distribution

Clr
Clear

CIDR
Classless Inter-Domain Routing

Control Plane
The control plane is concerned with the routing of packets and with providing a start or end point.

Core
A core may include several lcores or threads if the processor supports hyperthreading.

Core Components
A set of libraries provided by the DPDK, including eal, ring, mempool, mbuf, timers, and so on.

CPU
Central Processing Unit

CRC
Cyclic Redundancy Check

Data Plane
In contrast to the control plane, the data plane in a network architecture are the layers involved when forwarding packets. These layers must be highly optimized to achieve good performance.

DIMM
Dual In-line Memory Module

Doxygen
A documentation generator used in the DPDK to generate the API reference.

DPDK
Data Plane Development Kit

DRAM
Dynamic Random Access Memory

EAL
The Environment Abstraction Layer (EAL) provides a generic interface that hides the environment specifics from the applications and libraries. The services expected from the EAL are: development kit loading and launching, core affinity/ assignment procedures, system memory allocation/description, PCI bus access, inter-partition communication.

FIFO
First In First Out

FPGA
Field Programmable Gate Array

GbE
Gigabit Ethernet

HW
Hardware

HPET
High Precision Event Timer; a hardware timer that provides a precise time reference on x86 platforms.

ID
Identifier

IOCTL
Input/Output Control

I/O
Input/Output

IP
Internet Protocol

IPv4
Internet Protocol version 4

IPv6
Internet Protocol version 6

lcore
A logical execution unit of the processor, sometimes called a *hardware thread*.

KNI
Kernel Network Interface

L1
Layer 1

L2
Layer 2

L3
Layer 3

L4
Layer 4

LAN
Local Area Network

LPM
Longest Prefix Match

master lcore
The execution unit that executes the main() function and that launches other lcores.

mbuf
An mbuf is a data structure used internally to carry messages (mainly network packets). The name is derived from BSD stacks. To understand the concepts of packet buffers or mbuf, refer to *TCP/IP Illustrated, Volume 2: The Implementation*.

MESI
Modified Exclusive Shared Invalid (CPU cache coherency protocol)

MTU
Maximum Transfer Unit

NIC
Network Interface Card

OOO
Out Of Order (execution of instructions within the CPU pipeline)

NUMA
Non-uniform Memory Access

PCI
Peripheral Connect Interface

PHY
An abbreviation for the physical layer of the OSI model.

pktmbuf
An *mbuf* carrying a network packet.

PMD
Poll Mode Driver

QoS
Quality of Service

RCU
Read-Copy-Update algorithm, an alternative to simple rwlocks.

Rd
Read

RED
Random Early Detection

RSS
Receive Side Scaling

RTE
Run Time Environment. Provides a fast and simple framework for fast packet processing, in a lightweight environment as a Linux* application and using Poll Mode Drivers (PMDs) to increase speed.

Rx
Reception

Slave lcore
Any *lcore* that is not the *master lcore*.

Socket
A physical CPU, that includes several *cores*.

SLA
Service Level Agreement

srTCM
Single Rate Three Color Marking

SRTD
Scheduler Round Trip Delay

SW
Software

Target
In the DPDK, the target is a combination of architecture, machine, executive environment and toolchain. For example: i686-native-linux-gcc.

TCP
Transmission Control Protocol

TC
Traffic Class

TLB
Translation Lookaside Buffer

TLS
Thread Local Storage

trTCM
Two Rate Three Color Marking

TSC
Time Stamp Counter

Tx
Transmission

TUN/TAP
TUN and TAP are virtual network kernel devices.

VLAN
Virtual Local Area Network

Wr
Write

WRED
Weighted Random Early Detection

WRR
Weighted Round Robin

CHAPTER

SIX

HOWTO GUIDES

6.1 Live Migration of VM with SR-IOV VF

6.1.1 Overview

It is not possible to migrate a Virtual Machine which has an SR-IOV Virtual Function (VF).

To get around this problem the bonding PMD is used.

The following sections show an example of how to do this.

6.1.2 Test Setup

A bonded device is created in the VM. The virtio and VF PMD's are added as slaves to the bonded device. The VF is set as the primary slave of the bonded device.

A bridge must be set up on the Host connecting the tap device, which is the backend of the Virtio device and the Physical Function (PF) device.

To test the Live Migration two servers with identical operating systems installed are used. KVM and Qemu 2.3 is also required on the servers.

In this example, the servers have Niantic and or Fortville NIC's installed. The NIC's on both servers are connected to a switch which is also connected to the traffic generator.

The switch is configured to broadcast traffic on all the NIC ports. A *Sample switch configuration* can be found in this section.

The host is running the Kernel PF driver (ixgbe or i40e).

The ip address of host_server_1 is 10.237.212.46

The ip address of host_server_2 is 10.237.212.131

6.1.3 Live Migration steps

The sample scripts mentioned in the steps below can be found in the *Sample host scripts* and *Sample VM scripts* sections.

On host_server_1: Terminal 1

```
cd /root/dpdk/host_scripts
./setup_vf_on_212_46.sh
```

For Fortville NIC

```
./vm_virtio_vf_i40e_212_46.sh
```

For Niantic NIC

```
./vm_virtio_vf_one_212_46.sh
```

On host_server_1: Terminal 2

```
cd /root/dpdk/host_scripts
./setup_bridge_on_212_46.sh
./connect_to_qemu_mon_on_host.sh
(qemu)
```

On host_server_1: Terminal 1

In VM on host_server_1:

```
cd /root/dpdk/vm_scripts
./setup_dpdk_in_vm.sh
./run_testpmd_bonding_in_vm.sh

testpmd> show port info all
```

The `mac_addr` command only works with kernel PF for Niantic

```
testpmd> mac_addr add port 1 vf 0 AA:BB:CC:DD:EE:FF
```

The syntax of the `testpmd` command is:

Create bonded device (mode) (socket).

Mode 1 is active backup.

Virtio is port 0 (P0).

VF is port 1 (P1).

Bonding is port 2 (P2).

```
testpmd> create bonded device 1 0
Created new bonded device net_bond_testpmd_0 on (port 2).
testpmd> add bonding slave 0 2
testpmd> add bonding slave 1 2
testpmd> show bonding config 2
```

The syntax of the `testpmd` command is:

set bonding primary (slave id) (port id)

Set primary to P1 before starting bonding port.

```
testpmd> set bonding primary 1 2
testpmd> show bonding config 2
testpmd> port start 2
Port 2: 02:09:C0:68:99:A5
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex

testpmd> show bonding config 2
```

Primary is now P1. There are 2 active slaves.

Use P2 only for forwarding.

```
testpmd> set portlist 2
testpmd> show config fwd
testpmd> set fwd mac
testpmd> start
testpmd> show bonding config 2
```

Primary is now P1. There are 2 active slaves.

```
testpmd> show port stats all
```

VF traffic is seen at P1 and P2.

```
testpmd> clear port stats all
testpmd> set bonding primary 0 2
testpmd> remove bonding slave 1 2
testpmd> show bonding config 2
```

Primary is now P0. There is 1 active slave.

```
testpmd> clear port stats all
testpmd> show port stats all
```

No VF traffic is seen at P0 and P2, VF MAC address still present.

```
testpmd> port stop 1
testpmd> port close 1
```

Port close should remove VF MAC address, it does not remove perm_addr.

The `mac_addr` command only works with the kernel PF for Niantic.

```
testpmd> mac_addr remove 1 AA:BB:CC:DD:EE:FF
testpmd> port detach 1
Port '0000:00:04.0' is detached. Now total ports is 2
testpmd> show port stats all
```

No VF traffic is seen at P0 and P2.

On host_server_1: Terminal 2

```
(qemu) device_del vf1
```

On host_server_1: Terminal 1

In VM on host_server_1:

```
testpmd> show bonding config 2
```

Primary is now P0. There is 1 active slave.

```
testpmd> show port info all
testpmd> show port stats all
```

On host_server_2: Terminal 1

```
cd /root/dpdk/host_scripts
./setup_vf_on_212_131.sh
./vm_virtio_one_migrate.sh
```

On host_server_2: Terminal 2

```
./setup_bridge_on_212_131.sh
./connect_to_qemu_mon_on_host.sh
(qemu) info status
VM status: paused (inmigrate)
(qemu)
```

On host_server_1: Terminal 2

Check that the switch is up before migrating.

```
(qemu) migrate tcp:10.237.212.131:5555
(qemu) info status
VM status: paused (postmigrate)
```

For the Niantic NIC.

```
(qemu) info migrate
capabilities: xbzrle: off rdma-pin-all: off auto-converge: off zero-blocks: off
Migration status: completed
total time: 11834 milliseconds
downtime: 18 milliseconds
setup: 3 milliseconds
transferred ram: 389137 kbytes
throughput: 269.49 mbps
remaining ram: 0 kbytes
total ram: 1590088 kbytes
duplicate: 301620 pages
skipped: 0 pages
normal: 96433 pages
normal bytes: 385732 kbytes
dirty sync count: 2
(qemu) quit
```

For the Fortville NIC.

```
(qemu) info migrate
capabilities: xbzrle: off rdma-pin-all: off auto-converge: off zero-blocks: off
Migration status: completed
total time: 11619 milliseconds
downtime: 5 milliseconds
setup: 7 milliseconds
transferred ram: 379699 kbytes
throughput: 267.82 mbps
remaining ram: 0 kbytes
total ram: 1590088 kbytes
duplicate: 303985 pages
skipped: 0 pages
normal: 94073 pages
normal bytes: 376292 kbytes
dirty sync count: 2
(qemu) quit
```

On host_server_2: Terminal 1

In VM on host_server_2:

Hit Enter key. This brings the user to the testpmd prompt.

```
testpmd>
```

On host_server_2: Terminal 2

```
(qemu) info status
VM status: running
```

For the Niantic NIC.

```
(qemu) device_add pci-assign,host=06:10.0,id=vf1
```

For the Fortville NIC.

```
(qemu) device_add pci-assign,host=03:02.0,id=vf1
```

On host_server_2: Terminal 1

In VM on host_server_2:

```
testpmd> show port info all
testpmd> show port stats all
testpmd> show bonding config 2
testpmd> port attach 0000:00:04.0
Port 1 is attached.
Now total ports is 3
Done

testpmd> port start 1
```

The `mac_addr` command only works with the Kernel PF for Niantic.

```
testpmd> mac_addr add port 1 vf 0 AA:BB:CC:DD:EE:FF
testpmd> show port stats all.
testpmd> show config fwd
testpmd> show bonding config 2
testpmd> add bonding slave 1 2
testpmd> set bonding primary 1 2
testpmd> show bonding config 2
testpmd> show port stats all
```

VF traffic is seen at P1 (VF) and P2 (Bonded device).

```
testpmd> remove bonding slave 0 2
testpmd> show bonding config 2
testpmd> port stop 0
testpmd> port close 0
testpmd> port detach 0
Port '0000:00:03.0' is detached. Now total ports is 2

testpmd> show port info all
testpmd> show config fwd
testpmd> show port stats all
```

VF traffic is seen at P1 (VF) and P2 (Bonded device).

6.1.4 Sample host scripts

setup_vf_on_212_46.sh

Set up Virtual Functions on host_server_1

```
#!/bin/sh
# This script is run on the host 10.237.212.46 to setup the VF

# set up Niantic VF
cat /sys/bus/pci/devices/0000\:09\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:09\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:09\:00.0/sriov_numvfs
rmmod ixgbevf
```

(continues on next page)

(continued from previous page)

```
# set up Fortville VF
cat /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
rmmod i40evf
```

vm_virtio_vf_one_212_46.sh

Setup Virtual Machine on host_server_1

```
#!/bin/sh

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/username/disk_image/virt1_sml.disk"

# Number of guest cpus
VCPUS_NR="4"

# Memory
MEM=1536

taskset -c 1-5 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
 -vnc none -nographic \
 -hda $DISK_IMG \
 -netdev type=tap,id=net1,script=no,downscript=no,ifname=tap1 \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB \
 -device pci-assign,host=09:10.0,id=vf1 \
 -monitor telnet::3333,server,nowait
```

setup_bridge_on_212_46.sh

Setup bridge on host_server_1

```
#!/bin/sh
# This script is run on the host 10.237.212.46 to setup the bridge
# for the Tap device and the PF device.
# This enables traffic to go from the PF to the Tap to the Virtio PMD in the VM.

# ens3f0 is the Niantic NIC
# ens6f0 is the Fortville NIC

ifconfig ens3f0 down
ifconfig tap1 down
ifconfig ens6f0 down
ifconfig virbr0 down
```

(continues on next page)

(continued from previous page)

```
brctl show virbr0
brctl addif virbr0 ens3f0
brctl addif virbr0 ens6f0
brctl addif virbr0 tap1
brctl show virbr0

ifconfig ens3f0 up
ifconfig tap1 up
ifconfig ens6f0 up
ifconfig virbr0 up
```

connect_to_qemu_mon_on_host.sh

```
#!/bin/sh
# This script is run on both hosts when the VM is up,
# to connect to the Qemu Monitor.

telnet 0 3333
```

setup_vf_on_212_131.sh

Set up Virtual Functions on host_server_2

```
#!/bin/sh
# This script is run on the host 10.237.212.131 to setup the VF

# set up Niantic VF
cat /sys/bus/pci/devices/0000\:06\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:06\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:06\:00.0/sriov_numvfs
rmmod ixgbevf

# set up Fortville VF
cat /sys/bus/pci/devices/0000\:03\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:03\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:03\:00.0/sriov_numvfs
rmmod i40evf
```

vm_virtio_one_migrate.sh

Setup Virtual Machine on host_server_2

```
#!/bin/sh
# Start the VM on host_server_2 with the same parameters except without the VF
# parameters, as the VM on host_server_1, in migration-listen mode
# (-incoming tcp:0:5555)

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/username/disk_image/virt1_sml.disk"
```

(continues on next page)

(continued from previous page)

```
# Number of guest cpus
VCPUS_NR="4"

# Memory
MEM=1536

taskset -c 1-5 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
 -vnc none -nographic \
 -hda $DISK_IMG \
 -netdev type=tap,id=net1,script=no,downscript=no,ifname=tap1 \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB \
 -incoming tcp:0:5555 \
 -monitor telnet::3333,server,nowait
```

setup_bridge_on_212_131.sh

Setup bridge on host_server_2

```
#!/bin/sh
# This script is run on the host to setup the bridge
# for the Tap device and the PF device.
# This enables traffic to go from the PF to the Tap to the Virtio PMD in the VM.

# ens4f0 is the Niantic NIC
# ens5f0 is the Fortville NIC

ifconfig ens4f0 down
ifconfig tap1 down
ifconfig ens5f0 down
ifconfig virbr0 down

brctl show virbr0
brctl addif virbr0 ens4f0
brctl addif virbr0 ens5f0
brctl addif virbr0 tap1
brctl show virbr0

ifconfig ens4f0 up
ifconfig tap1 up
ifconfig ens5f0 up
ifconfig virbr0 up
```

6.1.5 Sample VM scripts

setup_dpdk_in_vm.sh

Set up DPDK in the Virtual Machine

```
#!/bin/sh
# this script matches the vm_virtio_vf_one script
# virtio port is 03
# vf port is 04

cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages

ifconfig -a
/root/dpdk/usertools/dpdk-devbind.py --status

rmmod virtio-pci ixgbevf

modprobe uio
insmod /root/dpdk/x86_64-default-linux-gcc/kmod/igb_uio.ko

/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:03.0
/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:04.0

/root/dpdk/usertools/dpdk-devbind.py --status
```

run_testpmd_bonding_in_vm.sh

Run testpmd in the Virtual Machine.

```
#!/bin/sh
# Run testpmd in the VM

# The test system has 8 cpus (0-7), use cpus 2-7 for VM
# Use taskset -pc <core number> <thread_id>

# use for bonding of virtio and vf tests in VM

/root/dpdk/x86_64-default-linux-gcc/app/testpmd \
-l 0-3 -n 4 --socket-mem 350 --  --i --port-topology=chained
```

6.1.6 Sample switch configuration

The Intel switch is used to connect the traffic generator to the NIC's on host_server_1 and host_server_2.

In order to run the switch configuration two console windows are required.

Log in as root in both windows.

TestPointShared, run_switch.sh and load /root/switch_config must be executed in the sequence below.

On Switch: Terminal 1

run TestPointShared

```
/usr/bin/TestPointShared
```

On Switch: Terminal 2

execute run_switch.sh

```
/root/run_switch.sh
```

On Switch: Terminal 1

load switch configuration

```
load /root/switch_config
```

Sample switch configuration script

The `/root/switch_config` script:

```
# TestPoint History
show port 1,5,9,13,17,21,25
set port 1,5,9,13,17,21,25 up
show port 1,5,9,13,17,21,25
del acl 1
create acl 1
create acl-port-set
create acl-port-set
add port port-set 1 0
add port port-set 5,9,13,17,21,25 1
create acl-rule 1 1
add acl-rule condition 1 1 port-set 1
add acl-rule action 1 1 redirect 1
apply acl
create vlan 1000
add vlan port 1000 1,5,9,13,17,21,25
set vlan tagging 1000 1,5,9,13,17,21,25 tag
set switch config flood_ucast fwd
show port stats all 1,5,9,13,17,21,25
```

6.2 Live Migration of VM with Virtio on host running vhost_user

6.2.1 Overview

Live Migration of a VM with DPDK Virtio PMD on a host which is running the Vhost sample application (vhost-switch) and using the DPDK PMD (ixgbe or i40e).

The Vhost sample application uses VMDQ so SRIOV must be disabled on the NIC's.

The following sections show an example of how to do this migration.

6.2.2 Test Setup

To test the Live Migration two servers with identical operating systems installed are used. KVM and QEMU is also required on the servers.

QEMU 2.5 is required for Live Migration of a VM with vhost_user running on the hosts.

In this example, the servers have Niantic and or Fortville NIC's installed. The NIC's on both servers are connected to a switch which is also connected to the traffic generator.

The switch is configured to broadcast traffic on all the NIC ports.

The ip address of host_server_1 is 10.237.212.46

The ip address of host_server_2 is 10.237.212.131

6.2.3 Live Migration steps

The sample scripts mentioned in the steps below can be found in the *Sample host scripts* and *Sample VM scripts* sections.

On host_server_1: Terminal 1

Setup DPDK on host_server_1

```
cd /root/dpdk/host_scripts
./setup_dpdk_on_host.sh
```

On host_server_1: Terminal 2

Bind the Niantic or Fortville NIC to igb_uio on host_server_1.

For Fortville NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:02:00.0
```

For Niantic NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:09:00.0
```

On host_server_1: Terminal 3

For Fortville and Niantic NIC's reset SRIOV and run the vhost_user sample application (vhost-switch) on host_server_1.

```
cd /root/dpdk/host_scripts
./reset_vf_on_212_46.sh
./run_vhost_switch_on_host.sh
```

On host_server_1: Terminal 1

Start the VM on host_server_1

```
./vm_virtio_vhost_user.sh
```

On host_server_1: Terminal 4

Connect to the QEMU monitor on host_server_1.

```
cd /root/dpdk/host_scripts
./connect_to_qemu_mon_on_host.sh
(qemu)
```

On host_server_1: Terminal 1

In VM on host_server_1:

Setup DPDK in the VM and run testpmd in the VM.

```
cd /root/dpdk/vm_scripts
./setup_dpdk_in_vm.sh
./run_testpmd_in_vm.sh

testpmd> show port info all
testpmd> set fwd mac retry
testpmd> start tx_first
testpmd> show port stats all
```

Virtio traffic is seen at P1 and P2.

On host_server_2: Terminal 1

Set up DPDK on the host_server_2.

```
cd /root/dpdk/host_scripts
./setup_dpdk_on_host.sh
```

On host_server_2: Terminal 2

Bind the Niantic or Fortville NIC to igb_uio on host_server_2.

For Fortville NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:03:00.0
```

For Niantic NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:06:00.0
```

On host_server_2: Terminal 3

For Fortville and Niantic NIC's reset SRIOV, and run the vhost_user sample application on host_server_2.

```
cd /root/dpdk/host_scripts
./reset_vf_on_212_131.sh
./run_vhost_switch_on_host.sh
```

On host_server_2: Terminal 1

Start the VM on host_server_2.

```
./vm_virtio_vhost_user_migrate.sh
```

On host_server_2: Terminal 4

Connect to the QEMU monitor on host_server_2.

```
cd /root/dpdk/host_scripts
./connect_to_qemu_mon_on_host.sh
(qemu) info status
VM status: paused (inmigrate)
(qemu)
```

On host_server_1: Terminal 4

Check that switch is up before migrating the VM.

```
(qemu) migrate tcp:10.237.212.131:5555
(qemu) info status
VM status: paused (postmigrate)

(qemu) info migrate
capabilities: xbzrle: off rdma-pin-all: off auto-converge: off zero-blocks: off
Migration status: completed
total time: 11619 milliseconds
downtime: 5 milliseconds
setup: 7 milliseconds
transferred ram: 379699 kbytes
throughput: 267.82 mbps
remaining ram: 0 kbytes
total ram: 1590088 kbytes
duplicate: 303985 pages
skipped: 0 pages
normal: 94073 pages
normal bytes: 376292 kbytes
dirty sync count: 2
(qemu) quit
```

On host_server_2: Terminal 1

In VM on host_server_2:

Hit Enter key. This brings the user to the testpmd prompt.

```
testpmd>
```

On host_server_2: Terminal 4

In QEMU monitor on host_server_2

```
(qemu) info status
VM status: running
```

On host_server_2: Terminal 1

In VM on host_server_2:

```
testpmd> show port info all
testpmd> show port stats all
```

Virtio traffic is seen at P0 and P1.

6.2.4 Sample host scripts

reset_vf_on_212_46.sh

```
#!/bin/sh
# This script is run on the host 10.237.212.46 to reset SRIOV

# BDF for Fortville NIC is 0000:02:00.0
cat /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:02\:00.0/max_vfs

# BDF for Niantic NIC is 0000:09:00.0
cat /sys/bus/pci/devices/0000\:09\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:09\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:09\:00.0/max_vfs
```

vm_virtio_vhost_user.sh

```
#/bin/sh
# Script for use with vhost_user sample application
# The host system has 8 cpu's (0-7)

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/user/disk_image/virt1_sml.disk"
```

(continues on next page)

(continued from previous page)

```
# Number of guest cpus
VCPUS_NR="6"

# Memory
MEM=1024

VIRTIO_OPTIONS="csum=off,gso=off,guest_tso4=off,guest_tso6=off,guest_ecn=off"

# Socket Path
SOCKET_PATH="/root/dpdk/host_scripts/usvhost"

taskset -c 2-7 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -object memory-backend-file,id=mem,size=1024M,mem-path=/mnt/huge,share=on \
 -numa node,memdev=mem,nodeid=0 \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
 -vnc none \
 -nographic \
 -hda $DISK_IMG \
 -chardev socket,id=chr0,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net1,chardev=chr0,vhostforce \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -chardev socket,id=chr1,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net2,chardev=chr1,vhostforce \
 -device virtio-net-pci,netdev=net2,mac=DD:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -monitor telnet::3333,server,nowait
```

connect_to_qemu_mon_on_host.sh

```
#!/bin/sh
# This script is run on both hosts when the VM is up,
# to connect to the Qemu Monitor.

telnet 0 3333
```

reset_vf_on_212_131.sh

```
#!/bin/sh
# This script is run on the host 10.237.212.131 to reset SRIOV

# BDF for Niantic NIC is 0000:06:00.0
cat /sys/bus/pci/devices/0000\:06\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:06\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:06\:00.0/max_vfs

# BDF for Fortville NIC is 0000:03:00.0
cat /sys/bus/pci/devices/0000\:03\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:03\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:03\:00.0/max_vfs
```

vm_virtio_vhost_user_migrate.sh

```
#/bin/sh
# Script for use with vhost user sample application
# The host system has 8 cpu's (0-7)

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/user/disk_image/virt1_sml.disk"

# Number of guest cpus
VCPUS_NR="6"

# Memory
MEM=1024

VIRTIO_OPTIONS="csum=off,gso=off,guest_tso4=off,guest_tso6=off,guest_ecn=off"

# Socket Path
SOCKET_PATH="/root/dpdk/host_scripts/usvhost"

taskset -c 2-7 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -object memory-backend-file,id=mem,size=1024M,mem-path=/mnt/huge,share=on \
 -numa node,memdev=mem,nodeid=0 \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
 -vnc none \
 -nographic \
 -hda $DISK_IMG \
 -chardev socket,id=chr0,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net1,chardev=chr0,vhostforce \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -chardev socket,id=chr1,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net2,chardev=chr1,vhostforce \
 -device virtio-net-pci,netdev=net2,mac=DD:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -incoming tcp:0:5555 \
 -monitor telnet::3333,server,nowait
```

6.2.5 Sample VM scripts

setup_dpdk_virtio_in_vm.sh

```
#!/bin/sh
# this script matches the vm_virtio_vhost_user script
# virtio port is 03
# virtio port is 04

cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages

ifconfig -a
```

(continues on next page)

(continued from previous page)

```
/root/dpdk/usertools/dpdk-devbind.py --status

rmmod virtio-pci

modprobe uio
insmod /root/dpdk/x86_64-default-linux-gcc/kmod/igb_uio.ko

/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:03.0
/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:04.0

/root/dpdk/usertools/dpdk-devbind.py --status
```

run_testpmd_in_vm.sh

```
#!/bin/sh
# Run testpmd for use with vhost_user sample app.
# test system has 8 cpus (0-7), use cpus 2-7 for VM

/root/dpdk/x86_64-default-linux-gcc/app/testpmd \
-l 0-5 -n 4 --socket-mem 350 -- --burst=64 --i
```

6.3 Flow Bifurcation How-to Guide

Flow Bifurcation is a mechanism which uses hardware capable Ethernet devices to split traffic between Linux user space and kernel space. Since it is a hardware assisted feature this approach can provide line rate processing capability. Other than *KNI*, the software is just required to enable device configuration, there is no need to take care of the packet movement during the traffic split. This can yield better performance with less CPU overhead.

The Flow Bifurcation splits the incoming data traffic to user space applications (such as DPDK applications) and/or kernel space programs (such as the Linux kernel stack). It can direct some traffic, for example data plane traffic, to DPDK, while directing some other traffic, for example control plane traffic, to the traditional Linux networking stack.

There are a number of technical options to achieve this. A typical example is to combine the technology of SR-IOV and packet classification filtering.

SR-IOV is a PCI standard that allows the same physical adapter to be split as multiple virtual functions. Each virtual function (VF) has separated queues with physical functions (PF). The network adapter will direct traffic to a virtual function with a matching destination MAC address. In a sense, SR-IOV has the capability for queue division.

Packet classification filtering is a hardware capability available on most network adapters. Filters can be configured to direct specific flows to a given receive queue by hardware. Different NICs may have different filter types to direct flows to a Virtual Function or a queue that belong to it.

In this way the Linux networking stack can receive specific traffic through the kernel driver while a DPDK application can receive specific traffic bypassing the Linux kernel by using drivers like VFIO or the DPDK `igb_uio` module.

Fig. 6.1: Flow Bifurcation Overview

6.3.1 Using Flow Bifurcation on Mellanox ConnectX

The Mellanox devices are *natively bifurcated*, so there is no need to split into SR-IOV PF/VF in order to get the flow bifurcation mechanism. The full device is already shared with the kernel driver.

The DPDK application can setup some flow steering rules, and let the rest go to the kernel stack. In order to define the filters strictly with flow rules, the *Flow isolated mode* can be configured.

There is no specific instructions to follow. The recommended reading is the *Generic flow API (rte_flow)* guide. Below is an example of testpmd commands for receiving VXLAN 42 in 4 queues of the DPDK port 0, while all other packets go to the kernel:

```
testpmd> flow isolate 0 true
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / vxlan vni is 42 / end \
         actions rss queues 0 1 2 3 end / end
```

6.4 Generic flow API - examples

This document demonstrates some concrete examples for programming flow rules with the `rte_flow` APIs.

- Detail of the rte_flow APIs can be found in the following link: *Generic flow API (rte_flow)*.
- Details of the TestPMD commands to set the flow rules can be found in the following link: *TestPMD Flow rules*

6.4.1 Simple IPv4 drop

Description

In this example we will create a simple rule that drops packets whose IPv4 destination equals 192.168.3.2. This code is equivalent to the following testpmd command (wrapped for clarity):

```
testpmd> flow create 0 ingress pattern eth / vlan /
                  ipv4 dst is 192.168.3.2 / end actions drop / end
```

Code

```
/* create the attribute structure */
struct rte_flow_attr attr = { .ingress = 1 };
struct rte_flow_item pattern[MAX_PATTERN_IN_FLOW];
struct rte_flow_action actions[MAX_ACTIONS_IN_FLOW];
struct rte_flow_item_eth eth;
struct rte_flow_item_vlan vlan;
struct rte_flow_item_ipv4 ipv4;
struct rte_flow *flow;
struct rte_flow_error error;

/* setting the eth to pass all packets */
pattern[0].type = RTE_FLOW_ITEM_TYPE_ETH;
pattern[0].spec = &eth;

/* set the vlan to pass all packets */
```

(continues on next page)

(continued from previous page)

```
pattern[1] = RTE_FLOW_ITEM_TYPE_VLAN;
pattern[1].spec = &vlan;

/* set the dst ipv4 packet to the required value */
ipv4.hdr.dst_addr = htonl(0xc0a80302);
pattern[2].type = RTE_FLOW_ITEM_TYPE_IPV4;
pattern[2].spec = &ipv4;

/* end the pattern array */
pattern[3].type = RTE_FLOW_ITEM_TYPE_END;

/* create the drop action */
actions[0].type = RTE_FLOW_ACTION_TYPE_DROP;
actions[1].type = RTE_FLOW_ACTION_TYPE_END;

/* validate and create the flow rule */
if (!rte_flow_validate(port_id, &attr, pattern, actions, &error))
    flow = rte_flow_create(port_id, &attr, pattern, actions, &error);
```

Output

Terminal 1: running sample app with the flow rule disabled:

```
./filter-program disable
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.4', dst='192.168.3.1'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.5', dst='192.168.3.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.4
received packet with src ip = 176.80.50.5
```

Terminal 1: running sample the app flow rule enabled:

```
./filter-program enabled
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.4', dst='192.168.3.1'),  \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.5', dst ='192.168.3.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.4
```

6.4.2 Range IPv4 drop

Description

In this example we will create a simple rule that drops packets whose IPv4 destination is in the range 192.168.3.0 to 192.168.3.255. This is done using a mask.

This code is equivalent to the following testpmd command (wrapped for clarity):

```
testpmd> flow create 0 ingress pattern eth / vlan /
                  ipv4 dst spec 192.168.3.0 dst mask 255.255.255.0 /
                  end actions drop / end
```

Code

```
struct rte_flow_attr attr = {.ingress = 1};
struct rte_flow_item pattern[MAX_PATTERN_IN_FLOW];
struct rte_flow_action actions[MAX_ACTIONS_IN_FLOW];
struct rte_flow_item_eth eth;
struct rte_flow_item_vlan vlan;
struct rte_flow_item_ipv4 ipv4;
struct rte_flow_item_ipv4 ipv4_mask;
struct rte_flow *flow;
struct rte_flow_error error;

/* setting the eth to pass all packets */
pattern[0].type = RTE_FLOW_ITEM_TYPE_ETH;
pattern[0].spec = &eth;

/* set the vlan to pass all packets */
pattern[1] = RTE_FLOW_ITEM_TYPE_VLAN;
pattern[1].spec = &vlan;

/* set the dst ipv4 packet to the required value */
ipv4.hdr.dst_addr = htonl(0xc0a80300);
ipv4_mask.hdr.dst_addr = htonl(0xffffff00);
pattern[2].type = RTE_FLOW_ITEM_TYPE_IPV4;
pattern[2].spec = &ipv4;
pattern[2].mask = &ipv4_mask;

/* end the pattern array */
pattern[3].type = RTE_FLOW_ITEM_TYPE_END;

/* create the drop action */
actions[0].type = RTE_FLOW_ACTION_TYPE_DROP;
actions[1].type = RTE_FLOW_ACTION_TYPE_END;

/* validate and create the flow rule */
if (!rte_flow_validate(port_id, &attr, pattern, actions, &error))
    flow = rte_flow_create(port_id, &attr, pattern, actions, &error);
```

Output

Terminal 1: running sample app flow rule disabled:

```
./filter-program disable
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.4', dst='192.168.3.1'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.5', dst='192.168.3.2'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.6', dst='192.168.5.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.4
received packet with src ip = 176.80.50.5
received packet with src ip = 176.80.50.6
```

Terminal 1: running sample app flow rule enabled:

```
./filter-program enabled
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.4', dst='192.168.3.1'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.5', dst='192.168.3.2'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.6', dst='192.168.5.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.6
```

6.4.3 Send vlan to queue

Description

In this example we will create a rule that routes all vlan id 123 to queue 3.

This code is equivalent to the following testpmd command (wrapped for clarity):

```
testpmd> flow create 0 ingress pattern eth / vlan vid spec 123 /
                  end actions queue index 3 / end
```

Code

```
struct rte_flow_attr attr = { .ingress = 1 };
struct rte_flow_item pattern[MAX_PATTERN_IN_FLOW];
struct rte_flow_action actions[MAX_ACTIONS_IN_FLOW];
struct rte_flow_item_eth eth;
struct rte_flow_item_vlan vlan;
struct rte_flow_action_queue queue = { .index = 3 };
struct rte_flow *flow;
struct rte_flow_error error;

/* setting the eth to pass all packets */
pattern[0].type = RTE_FLOW_ITEM_TYPE_ETH;
pattern[0].spec = &eth;

/* set the vlan to pas all packets */
vlan.vid = 123;
pattern[1] = RTE_FLOW_ITEM_TYPE_VLAN;
pattern[1].spec = &vlan;

/* end the pattern array */
pattern[2].type = RTE_FLOW_ITEM_TYPE_END;

/* create the queue action */
actions[0].type = RTE_FLOW_ACTION_TYPE_QUEUE;
actions[0].conf = &queue;
actions[1].type = RTE_FLOW_ACTION_TYPE_END;

/* validate and create the flow rule */
if (!rte_flow_validate(port_id, &attr, pattern, actions, &error))
    flow = rte_flow_create(port_id, &attr, pattern, actions, &error);
```

Output

Terminal 1: running sample app flow rule disabled:

```
./filter-program disable
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q(vlan=123)/IP(src='176.80.50.4', dst='192.168.3.1'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q(vlan=50)/IP(src='176.80.50.5', dst='192.168.3.2'),  \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q(vlan=123)/IP(src='176.80.50.6', dst='192.168.5.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.4 sent to queue 2
received packet with src ip = 176.80.50.5 sent to queue 1
received packet with src ip = 176.80.50.6 sent to queue 0
```

Terminal 1: running sample app flow rule enabled:

```
./filter-program enabled
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q(vlan=123)/IP(src='176.80.50.4', dst='192.168.3.1'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q(vlan=50)/IP(src='176.80.50.5', dst='192.168.3.2'),  \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q(vlan=123)/IP(src='176.80.50.6', dst='192.168.5.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.4 sent to queue 3
received packet with src ip = 176.80.50.5 sent to queue 1
received packet with src ip = 176.80.50.6 sent to queue 3
```

6.5 PVP reference benchmark setup using testpmd

This guide lists the steps required to setup a PVP benchmark using testpmd as a simple forwarder between NICs and Vhost interfaces. The goal of this setup is to have a reference PVP benchmark without using external vSwitches (OVS, VPP, ...) to make it easier to obtain reproducible results and to facilitate continuous integration testing.

The guide covers two ways of launching the VM, either by directly calling the QEMU command line, or by relying on libvirt. It has been tested with DPDK v16.11 using RHEL7 for both host and guest.

6.5.1 Setup overview

Fig. 6.2: PVP setup using 2 NICs

In this diagram, each red arrow represents one logical core. This use-case requires 6 dedicated logical cores. A forwarding configuration with a single NIC is also possible, requiring 3 logical cores.

6.5.2 Host setup

In this setup, we isolate 6 cores (from CPU2 to CPU7) on the same NUMA node. Two cores are assigned to the VM vCPUs running testpmd and four are assigned to testpmd on the host.

Host tuning

1. On BIOS, disable turbo-boost and hyper-threads.
2. Append these options to Kernel command line:

```
intel_pstate=disable mce=ignore_ce default_hugepagesz=1G hugepagesz=1G hugepages=6␣
↪isolcpus=2-7 rcu_nocbs=2-7 nohz_full=2-7 iommu=pt intel_iommu=on
```

3. Disable hyper-threads at runtime if necessary or if BIOS is not accessible:

```
cat /sys/devices/system/cpu/cpu*[0-9]/topology/thread_siblings_list \
    | sort | uniq \
    | awk -F, '{system("echo 0 > /sys/devices/system/cpu/cpu"$2"/online")}'
```

4. Disable NMIs:

```
echo 0 > /proc/sys/kernel/nmi_watchdog
```

5. Exclude isolated CPUs from the writeback cpumask:

```
echo ffffff03 > /sys/bus/workqueue/devices/writeback/cpumask
```

6. Isolate CPUs from IRQs:

```
clear_mask=0xfc #Isolate CPU2 to CPU7 from IRQs
for i in /proc/irq/*/smp_affinity
do
  echo "obase=16;$(( 0x$(cat $i) & ~$clear_mask ))" | bc > $i
done
```

Qemu build

Build Qemu:

```
git clone git://git.qemu.org/qemu.git
cd qemu
mkdir bin
cd bin
../configure --target-list=x86_64-softmmu
make
```

DPDK build

Build DPDK:

```
git clone git://dpdk.org/dpdk
cd dpdk
export RTE_SDK=$PWD
make install T=x86_64-native-linux-gcc DESTDIR=install
```

Testpmd launch

1. Assign NICs to DPDK:

```
modprobe vfio-pci
$RTE_SDK/install/sbin/dpdk-devbind -b vfio-pci 0000:11:00.0 0000:11:00.1
```

Note: The Sandy Bridge family seems to have some IOMMU limitations giving poor performance results. To achieve good performance on these machines consider using UIO instead.

2. Launch the testpmd application:

```
$RTE_SDK/install/bin/testpmd -l 0,2,3,4,5 --socket-mem=1024 -n 4 \
    --vdev 'net_vhost0,iface=/tmp/vhost-user1' \
    --vdev 'net_vhost1,iface=/tmp/vhost-user2' -- \
    --portmask=f -i --rxq=1 --txq=1 \
    --nb-cores=4 --forward-mode=io
```

With this command, isolated CPUs 2 to 5 will be used as lcores for PMD threads.

3. In testpmd interactive mode, set the portlist to obtain the correct port chaining:

```
set portlist 0,2,1,3
start
```

VM launch

The VM may be launched either by calling QEMU directly, or by using libvirt.

Qemu way

Launch QEMU with two Virtio-net devices paired to the vhost-user sockets created by testpmd. Below example uses default Virtio-net options, but options may be specified, for example to disable mergeable buffers or indirect descriptors.

```
<QEMU path>/bin/x86_64-softmmu/qemu-system-x86_64 \
    -enable-kvm -cpu host -m 3072 -smp 3 \
    -chardev socket,id=char0,path=/tmp/vhost-user1 \
    -netdev type=vhost-user,id=mynet1,chardev=char0,vhostforce \
    -device virtio-net-pci,netdev=mynet1,mac=52:54:00:02:d9:01,addr=0x10 \
    -chardev socket,id=char1,path=/tmp/vhost-user2 \
    -netdev type=vhost-user,id=mynet2,chardev=char1,vhostforce \
    -device virtio-net-pci,netdev=mynet2,mac=52:54:00:02:d9:02,addr=0x11 \
    -object memory-backend-file,id=mem,size=3072M,mem-path=/dev/hugepages,share=on \
    -numa node,memdev=mem -mem-prealloc \
    -net user,hostfwd=tcp::1002$1-:22 -net nic \
    -qmp unix:/tmp/qmp.socket,server,nowait \
    -monitor stdio <vm_image>.qcow2
```

You can use this qmp-vcpu-pin script to pin vCPUs.

It can be used as follows, for example to pin 3 vCPUs to CPUs 1, 6 and 7, where isolated CPUs 6 and 7 will be used as lcores for Virtio PMDs:

```
export PYTHONPATH=$PYTHONPATH:<QEMU path>/scripts/qmp
./qmp-vcpu-pin -s /tmp/qmp.socket 1 6 7
```

Libvirt way

Some initial steps are required for libvirt to be able to connect to testpmd's sockets.

First, SELinux policy needs to be set to permissive, since testpmd is generally run as root (note, as reboot is required):

```
cat /etc/selinux/config

# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing  - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing.
#     disabled   - No SELinux policy is loaded.
SELINUX=permissive

# SELINUXTYPE= can take one of three two values:
#     targeted - Targeted processes are protected,
#     minimum  - Modification of targeted policy.
#                Only selected processes are protected.
#     mls      - Multi Level Security protection.
SELINUXTYPE=targeted
```

Also, Qemu needs to be run as root, which has to be specified in `/etc/libvirt/qemu.conf`:

```
user = "root"
```

Once the domain created, the following snippet is an extract of he most important information (hugepages, vCPU pinning, Virtio PCI devices):

```
<domain type='kvm'>
  <memory unit='KiB'>3145728</memory>
  <currentMemory unit='KiB'>3145728</currentMemory>
  <memoryBacking>
    <hugepages>
      <page size='1048576' unit='KiB' nodeset='0'/>
    </hugepages>
    <locked/>
  </memoryBacking>
  <vcpu placement='static'>3</vcpu>
  <cputune>
    <vcpupin vcpu='0' cpuset='1'/>
    <vcpupin vcpu='1' cpuset='6'/>
    <vcpupin vcpu='2' cpuset='7'/>
    <emulatorpin cpuset='0'/>
  </cputune>
  <numatune>
    <memory mode='strict' nodeset='0'/>
  </numatune>
  <os>
    <type arch='x86_64' machine='pc-i440fx-rhel7.0.0'>hvm</type>
    <boot dev='hd'/>
  </os>
  <cpu mode='host-passthrough'>
    <topology sockets='1' cores='3' threads='1'/>
```

(continues on next page)

(continued from previous page)

```
    <numa>
      <cell id='0' cpus='0-2' memory='3145728' unit='KiB' memAccess='shared'/>
    </numa>
  </cpu>
  <devices>
    <interface type='vhostuser'>
      <mac address='56:48:4f:53:54:01'/>
      <source type='unix' path='/tmp/vhost-user1' mode='client'/>
      <model type='virtio'/>
      <driver name='vhost' rx_queue_size='256' />
      <address type='pci' domain='0x0000' bus='0x00' slot='0x10' function='0x0'/>
    </interface>
    <interface type='vhostuser'>
      <mac address='56:48:4f:53:54:02'/>
      <source type='unix' path='/tmp/vhost-user2' mode='client'/>
      <model type='virtio'/>
      <driver name='vhost' rx_queue_size='256' />
      <address type='pci' domain='0x0000' bus='0x00' slot='0x11' function='0x0'/>
    </interface>
  </devices>
</domain>
```

6.5.3 Guest setup

Guest tuning

1. Append these options to the Kernel command line:

```
default_hugepagesz=1G hugepagesz=1G hugepages=1 intel_iommu=on iommu=pt isolcpus=1,2 rcu_
↪nocbs=1,2 nohz_full=1,2
```

2. Disable NMIs:

```
echo 0 > /proc/sys/kernel/nmi_watchdog
```

3. Exclude isolated CPU1 and CPU2 from the writeback cpumask:

```
echo 1 > /sys/bus/workqueue/devices/writeback/cpumask
```

4. Isolate CPUs from IRQs:

```
clear_mask=0x6 #Isolate CPU1 and CPU2 from IRQs
for i in /proc/irq/*/smp_affinity
do
  echo "obase=16;$(( 0x$(cat $i) & ~$clear_mask ))" | bc > $i
done
```

DPDK build

Build DPDK:

```
git clone git://dpdk.org/dpdk
cd dpdk
export RTE_SDK=$PWD
make install T=x86_64-native-linux-gcc DESTDIR=install
```

Testpmd launch

Probe vfio module without iommu:

```
modprobe -r vfio_iommu_type1
modprobe -r vfio
modprobe  vfio enable_unsafe_noiommu_mode=1
cat /sys/module/vfio/parameters/enable_unsafe_noiommu_mode
modprobe vfio-pci
```

Bind the virtio-net devices to DPDK:

```
$RTE_SDK/usertools/dpdk-devbind.py -b vfio-pci 0000:00:10.0 0000:00:11.0
```

Start testpmd:

```
$RTE_SDK/install/bin/testpmd -l 0,1,2 --socket-mem 1024 -n 4 \
    --proc-type auto --file-prefix pg -- \
    --portmask=3 --forward-mode=macswap --port-topology=chained \
    --disable-rss -i --rxq=1 --txq=1 \
    --rxd=256 --txd=256 --nb-cores=2 --auto-start
```

6.5.4 Results template

Below template should be used when sharing results:

```
Traffic Generator: <Test equipment (e.g. IXIA, Moongen, ...)>
Acceptable Loss: <n>%
Validation run time: <n>min
Host DPDK version/commit: <version, SHA-1>
Guest DPDK version/commit: <version, SHA-1>
Patches applied: <link to patchwork>
QEMU version/commit: <version>
Virtio features: <features (e.g. mrg_rxbuf='off', leave empty if default)>
CPU: <CPU model>, <CPU frequency>
NIC: <NIC model>
Result: <n> Mpps
```

6.6 VF daemon (VFd)

VFd (the VF daemon) is a mechanism which can be used to configure features on a VF (SR-IOV Virtual Function) without direct access to the PF (SR-IOV Physical Function). VFd is an *EXPERIMENTAL* feature which can only be used in the scenario of DPDK PF with a DPDK VF. If the PF port is driven by the Linux kernel driver then the VFd feature will not work. Currently VFd is only supported by the ixgbe and i40e drivers.

In general VF features cannot be configured directly by an end user application since they are under the control of the PF. The normal approach to configuring a feature on a VF is that an application would call the APIs provided by the VF driver. If the required feature cannot be configured by the VF directly (the most common case) the VF sends a message to the PF through the mailbox on ixgbe and i40e. This means that the availability of the feature depends on whether the appropriate mailbox messages are defined.

DPDK leverages the mailbox interface defined by the Linux kernel driver so that compatibility with the kernel driver can be guaranteed. The downside of this approach is that the availability of messages supported by the kernel become a limitation when the user wants to configure features on the VF.

VFd is a new method of controlling the features on a VF. The VF driver doesn't talk directly to the PF driver when configuring a feature on the VF. When a VF application (i.e., an application using the VF ports) wants to enable a VF feature, it can send a message to the PF application (i.e., the application using the PF port, which can be the same as the VF application). The PF application will configure the feature for the VF. Obviously, the PF application can also configure the VF features without a request from the VF application.

Fig. 6.3: VF daemon (VFd) Overview

Compared with the traditional approach the VFd moves the negotiation between VF and PF from the driver level to application level. So the application should define how the negotiation between the VF and PF works, or even if the control should be limited to the PF.

It is the application's responsibility to use VFd. Consider for example a KVM migration, the VF application may transfer from one VM to another. It is recommended in this case that the PF control the VF features without participation from the VF. Then the VF application has no capability to configure the features. So the user doesn't need to define the interface between the VF application and the PF application. The service provider should take the control of all the features.

The following sections describe the VFd functionality.

Note: Although VFd is supported by both ixgbe and i40e, please be aware that since the hardware capability is different, the functions supported by ixgbe and i40e are not the same.

6.6.1 Preparing

VFd only can be used in the scenario of DPDK PF + DPDK VF. Users should bind the PF port to `igb_uio`, then create the VFs based on the DPDK PF host.

The typical procedure to achieve this is as follows:

1. Boot the system without iommu, or with `iommu=pt`.
2. Bind the PF port to `igb_uio`, for example:

```
dpdk-devbind.py -b igb_uio 01:00.0
```

3. Create a Virtual Function:

```
echo 1 > /sys/bus/pci/devices/0000:01:00.0/max_vfs
```

4. Start a VM with the new VF port bypassed to it.
5. Run a DPDK application on the PF in the host:

```
testpmd -l 0-7 -n 4 -- -i --txqflags=0
```

6. Bind the VF port to `igb_uio` in the VM:

```
dpdk-devbind.py -b igb_uio 03:00.0
```

7. Run a DPDK application on the VF in the VM:

```
testpmd -l 0-7 -n 4 -- -i --txqflags=0
```

6.6.2 Common functions of IXGBE and I40E

The following sections show how to enable PF/VF functionality based on the above testpmd setup.

TX loopback

Run a testpmd runtime command on the PF to set TX loopback:

```
set tx loopback 0 on|off
```

This sets whether the PF port and all the VF ports that belong to it are allowed to send the packets to other virtual ports.

Although it is a VFd function, it is the global setting for the whole physical port. When using this function, the PF and all the VFs TX loopback will be enabled/disabled.

VF MAC address setting

Run a testpmd runtime command on the PF to set the MAC address for a VF port:

```
set vf mac addr 0 0 A0:36:9F:7B:C3:51
```

This testpmd runtime command will change the MAC address of the VF port to this new address. If any other addresses are set before, they will be overwritten.

VF MAC anti-spoofing

Run a testpmd runtime command on the PF to enable/disable the MAC anti-spoofing for a VF port:

```
set vf mac antispoof 0 0 on|off
```

When enabling the MAC anti-spoofing, the port will not forward packets whose source MAC address is not the same as the port.

VF VLAN anti-spoofing

Run a testpmd runtime command on the PF to enable/disable the VLAN anti-spoofing for a VF port:

```
set vf vlan antispoof 0 0 on|off
```

When enabling the VLAN anti-spoofing, the port will not send packets whose VLAN ID does not belong to VLAN IDs that this port can receive.

VF VLAN insertion

Run a testpmd runtime command on the PF to set the VLAN insertion for a VF port:

```
set vf vlan insert 0 0 1
```

When using this testpmd runtime command, an assigned VLAN ID can be inserted to the transmitted packets by the hardware.

The assigned VLAN ID can be 0. It means disabling the VLAN insertion.

VF VLAN stripping

Run a testpmd runtime command on the PF to enable/disable the VLAN stripping for a VF port:

```
set vf vlan stripq 0 0 on|off
```

This testpmd runtime command is used to enable/disable the RX VLAN stripping for a specific VF port.

VF VLAN filtering

Run a testpmd runtime command on the PF to set the VLAN filtering for a VF port:

```
rx_vlan add 1 port 0 vf 1
rx_vlan rm  1 port 0 vf 1
```

These two testpmd runtime commands can be used to add or remove the VLAN filter for several VF ports. When the VLAN filters are added only the packets that have the assigned VLAN IDs can be received. Other packets will be dropped by hardware.

6.6.3 The IXGBE specific VFd functions

The functions in this section are specific to the ixgbe driver.

All queues drop

Run a testpmd runtime command on the PF to enable/disable the all queues drop:

```
set all queues drop on|off
```

This is a global setting for the PF and all the VF ports of the physical port.

Enabling the `all queues drop` feature means that when there is no available descriptor for the received packets they are dropped. The `all queues drop` feature should be enabled in SR-IOV mode to avoid one queue blocking others.

VF packet drop

Run a testpmd runtime command on the PF to enable/disable the packet drop for a specific VF:

```
set vf split drop 0 0 on|off
```

This is a similar function as `all queues drop`. The difference is that this function is per VF setting and the previous function is a global setting.

VF rate limit

Run a testpmd runtime command on the PF to all queues' rate limit for a specific VF:

```
set port 0 vf 0 rate 10 queue_mask 1
```

This is a function to set the rate limit for all the queues in the `queue_mask` bitmap. It is not used to set the summary of the rate limit. The rate limit of every queue will be set equally to the assigned rate limit.

VF RX enabling

Run a testpmd runtime command on the PF to enable/disable packet receiving for a specific VF:

```
set port 0 vf 0 rx on|off
```

This function can be used to stop/start packet receiving on a VF.

VF TX enabling

Run a testpmd runtime command on the PF to enable/disable packet transmitting for a specific VF:

```
set port 0 vf 0 tx on|off
```

This function can be used to stop/start packet transmitting on a VF.

VF RX mode setting

Run a testpmd runtime command on the PF to set the RX mode for a specific VF:

```
set port 0 vf 0 rxmode AUPE|ROPE|BAM|MPE on|off
```

This function can be used to enable/disable some RX modes on the VF, including:

- If it accept untagged packets.
- If it accepts packets matching the MAC filters.
- If it accept MAC broadcast packets,
- If it enables MAC multicast promiscuous mode.

6.6.4 The I40E specific VFd functions

The functions in this section are specific to the i40e driver.

VF statistics

This provides an API to get the a specific VF's statistic from PF.

VF statistics resetting

This provides an API to rest the a specific VF's statistic from PF.

VF link status change notification

This provide an API to let a specific VF know if the physical link status changed.

Normally if a VF received this notification, the driver should notify the application to reset the VF port.

VF MAC broadcast setting

Run a testpmd runtime command on the PF to enable/disable MAC broadcast packet receiving for a specific VF:

```
set vf broadcast 0 0 on|off
```

VF MAC multicast promiscuous mode

Run a testpmd runtime command on the PF to enable/disable MAC multicast promiscuous mode for a specific VF:

```
set vf allmulti 0 0 on|off
```

VF MAC unicast promiscuous mode

Run a testpmd runtime command on the PF to enable/disable MAC unicast promiscuous mode for a specific VF:

```
set vf promisc 0 0 on|off
```

VF max bandwidth

Run a testpmd runtime command on the PF to set the TX maximum bandwidth for a specific VF:

```
set vf tx max-bandwidth 0 0 2000
```

The maximum bandwidth is an absolute value in Mbps.

VF TC bandwidth allocation

Run a testpmd runtime command on the PF to set the TCs (traffic class) TX bandwidth allocation for a specific VF:

```
set vf tc tx min-bandwidth 0 0 (20,20,20,40)
```

The allocated bandwidth should be set for all the TCs. The allocated bandwidth is a relative value as a percentage. The sum of all the bandwidth should be 100.

VF TC max bandwidth

Run a testpmd runtime command on the PF to set the TCs TX maximum bandwidth for a specific VF:

```
set vf tc tx max-bandwidth 0 0 0 10000
```

The maximum bandwidth is an absolute value in Mbps.

TC strict priority scheduling

Run a testpmd runtime command on the PF to enable/disable several TCs TX strict priority scheduling:

```
set tx strict-link-priority 0 0x3
```

The 0 in the TC bitmap means disabling the strict priority scheduling for this TC. To enable use a value of 1.

6.7 Virtio_user for Container Networking

Container becomes more and more popular for strengths, like low overhead, fast boot-up time, and easy to deploy, etc. How to use DPDK to accelerate container networking becomes a common question for users. There are two use models of running DPDK inside containers, as shown in Fig. 6.4.

Fig. 6.4: Use models of running DPDK inside container

This page will only cover aggregation model.

6.7.1 Overview

The virtual device, virtio-user, with unmodified vhost-user backend, is designed for high performance user space container networking or inter-process communication (IPC).

The overview of accelerating container networking by virtio-user is shown in Fig. 6.5.

Fig. 6.5: Overview of accelerating container networking by virtio-user

Different virtio PCI devices we usually use as a para-virtualization I/O in the context of QEMU/VM, the basic idea here is to present a kind of virtual devices, which can be attached and initialized by DPDK. The device emulation layer by QEMU in VM's context is saved by just registering a new kind of virtual device in DPDK's ether layer. And to minimize the change, we reuse already-existing virtio PMD code (driver/net/virtio/).

Virtio, in essence, is a shm-based solution to transmit/receive packets. How is memory shared? In VM's case, qemu always shares the whole physical layout of VM to vhost backend. But it's not feasible for a container, as a process, to share all virtual memory regions to backend. So only those virtual memory regions (aka, hugepages initialized in DPDK) are sent to backend. It restricts that only addresses in these areas can be used to transmit or receive packets.

6.7.2 Sample Usage

Here we use Docker as container engine. It also applies to LXC, Rocket with some minor changes.

1. Compile DPDK.

```
make install RTE_SDK=`pwd` T=x86_64-native-linux-gcc
```

2. Write a Dockerfile like below.

```
cat <<EOT >> Dockerfile
FROM ubuntu:latest
WORKDIR /usr/src/dpdk
COPY . /usr/src/dpdk
ENV PATH "$PATH:/usr/src/dpdk/x86_64-native-linux-gcc/app/"
EOT
```

3. Build a Docker image.

```
docker build -t dpdk-app-testpmd .
```

4. Start a testpmd on the host with a vhost-user port.

```
$(testpmd) -l 0-1 -n 4 --socket-mem 1024,1024 \
    --vdev 'eth_vhost0,iface=/tmp/sock0' \
    --file-prefix=host --no-pci -- -i
```

5. Start a container instance with a virtio-user port.

```
docker run -i -t -v /tmp/sock0:/var/run/usvhost \
    -v /dev/hugepages:/dev/hugepages \
    dpdk-app-testpmd testpmd -l 6-7 -n 4 -m 1024 --no-pci \
    --vdev=virtio_user0,path=/var/run/usvhost \
    --file-prefix=container \
    -- -i
```

Note: If we run all above setup on the host, it's a shm-based IPC.

6.7.3 Limitations

We have below limitations in this solution:

- Cannot work with –huge-unlink option. As we need to reopen the hugepage file to share with vhost backend.
- Cannot work with –no-huge option. Currently, DPDK uses anonymous mapping under this option which cannot be reopened to share with vhost backend.
- Cannot work when there are more than VHOST_MEMORY_MAX_NREGIONS(8) hugepages. If you have more regions (especially when 2MB hugepages are used), the option, –single-file-segments, can help to reduce the number of shared files.
- Applications should not use file name like HUGEFILE_FMT ("%smap_%d"). That will bring confusion when sharing hugepage files with backend by name.
- Root privilege is a must. DPDK resolves physical addresses of hugepages which seems not necessary, and some discussions are going on to remove this restriction.

6.8 Virtio_user as Exceptional Path

The virtual device, virtio-user, was originally introduced with vhost-user backend, as a high performance solution for IPC (Inter-Process Communication) and user space container networking.

Virtio_user with vhost-kernel backend is a solution for exceptional path, such as KNI which exchanges packets with kernel networking stack. This solution is very promising in:

- Maintenance

  All kernel modules needed by this solution, vhost and vhost-net (kernel), are upstreamed and extensively used kernel module.

- Features

  vhost-net is born to be a networking solution, which has lots of networking related features, like multi queue, tso, multi-seg mbuf, etc.

- Performance

  similar to KNI, this solution would use one or more kthreads to send/receive packets to/from user space DPDK applications, which has little impact on user space polling thread (except that it might enter into kernel space to wake up those kthreads if necessary).

The overview of an application using virtio-user as exceptional path is shown in Fig. 6.6.

Fig. 6.6: Overview of a DPDK app using virtio-user as exceptional path

6.8.1 Sample Usage

As a prerequisite, the vhost/vhost-net kernel CONFIG should be chosen before compiling the kernel and those kernel modules should be inserted.

1. Compile DPDK and bind a physical NIC to igb_uio/uio_pci_generic/vfio-pci.

   This physical NIC is for communicating with outside.

2. Run testpmd.

```
$(testpmd) -l 2-3 -n 4 \
        --vdev=virtio_user0,path=/dev/vhost-net,queue_size=1024 \
        -- -i --tx-offloads=0x0000002c --enable-lro \
        --txd=1024 --rxd=1024
```

   This command runs testpmd with two ports, one physical NIC to communicate with outside, and one virtio-user to communicate with kernel.

- `--enable-lro`

  This is used to negotiate VIRTIO_NET_F_GUEST_TSO4 and VIRTIO_NET_F_GUEST_TSO6 feature so that large packets from kernel can be transmitted to DPDK application and further TSOed by physical NIC.

- `queue_size`

  256 by default. To avoid shortage of descriptors, we can increase it to 1024.

- `queues`

Number of multi-queues. Each queue will be served by a kthread. For example:

```
$(testpmd) -l 2-3 -n 4 \
        --vdev=virtio_user0,path=/dev/vhost-net,queues=2,queue_size=1024 \
        -- -i --tx-offloads=0x0000002c --enable-lro \
        --txq=2 --rxq=2 --txd=1024 --rxd=1024
```

1. Enable Rx checksum offloads in testpmd:

```
(testpmd) port stop 0
(testpmd) port config 0 rx_offload tcp_cksum on
(testpmd) port config 0 rx_offload udp_cksum on
(testpmd) port start 0
```

2. Start testpmd:

```
(testpmd) start
```

3. Configure IP address and start tap:

```
ifconfig tap0 1.1.1.1/24 up
```

Note: The tap device will be named tap0, tap1, etc, by kernel.

Then, all traffic from physical NIC can be forwarded into kernel stack, and all traffic on the tap0 can be sent out from physical NIC.

6.8.2 Limitations

This solution is only available on Linux systems.

6.9 DPDK pdump Library and pdump Tool

This document describes how the Data Plane Development Kit (DPDK) Packet Capture Framework is used for capturing packets on DPDK ports. It is intended for users of DPDK who want to know more about the Packet Capture feature and for those who want to monitor traffic on DPDK-controlled devices.

The DPDK packet capture framework was introduced in DPDK v16.07. The DPDK packet capture framework consists of the DPDK pdump library and DPDK pdump tool.

6.9.1 Introduction

The *librte_pdump* library provides the APIs required to allow users to initialize the packet capture framework and to enable or disable packet capture. The library works on a client/server model and its usage is recommended for debugging purposes.

The *dpdk-pdump* tool is developed based on the `librte_pdump` library. It runs as a DPDK secondary process and is capable of enabling or disabling packet capture on DPDK ports. The `dpdk-pdump` tool provides command-line options with which users can request enabling or disabling of the packet capture on DPDK ports.

The application which initializes the packet capture framework will act as a server and the application that enables or disables the packet capture will act as a client. The server sends the Rx and Tx packets from the DPDK ports to the client.

In DPDK the `testpmd` application can be used to initialize the packet capture framework and act as a server, and the `dpdk-pdump` tool acts as a client. To view Rx or Tx packets of `testpmd`, the application should be launched first, and then the `dpdk-pdump` tool. Packets from `testpmd` will be sent to the tool, which then sends them on to the Pcap PMD device and that device writes them to the Pcap file or to an external interface depending on the command-line option used.

Some things to note:

- The `dpdk-pdump` tool can only be used in conjunction with a primary application which has the packet capture framework initialized already. In dpdk, only `testpmd` is modified to initialize packet capture framework, other applications remain untouched. So, if the `dpdk-pdump` tool has to be used with any application other than the testpmd, the user needs to explicitly modify that application to call the packet capture framework initialization code. Refer to the `app/test-pmd/testpmd.c` code and look for `pdump` keyword to see how this is done.
- The `dpdk-pdump` tool depends on the libpcap based PMD which is disabled by default in the build configuration files, owing to an external dependency on the libpcap development files. Once the libpcap development files are installed, the libpcap based PMD can be enabled by setting `CONFIG_RTE_LIBRTE_PMD_PCAP=y` and recompiling the DPDK.

6.9.2 Test Environment

The overview of using the Packet Capture Framework and the `dpdk-pdump` tool for packet capturing on the DPDK port in Fig. 6.7.

Fig. 6.7: Packet capturing on a DPDK port using the dpdk-pdump tool.

6.9.3 Configuration

Modify the DPDK primary application to initialize the packet capture framework as mentioned in the above notes and enable the following config options and build DPDK:

```
CONFIG_RTE_LIBRTE_PMD_PCAP=y
CONFIG_RTE_LIBRTE_PDUMP=y
```

6.9.4 Running the Application

The following steps demonstrate how to run the `dpdk-pdump` tool to capture Rx side packets on dpdk_port0 in Fig. 6.7 and inspect them using `tcpdump`.

1. Launch testpmd as the primary application:

```
sudo ./app/testpmd -c 0xf0 -n 4 -- -i --port-topology=chained
```

2. Launch the pdump tool as follows:

```
sudo ./build/app/dpdk-pdump -- \
     --pdump 'port=0,queue=*,rx-dev=/tmp/capture.pcap'
```

3. Send traffic to dpdk_port0 from traffic generator. Inspect packets captured in the file capture.pcap using a tool that can interpret Pcap files, for example tcpdump:

```
$tcpdump -nr /tmp/capture.pcap
reading from file /tmp/capture.pcap, link-type EN10MB (Ethernet)
11:11:36.891404 IP 4.4.4.4.whois++ > 3.3.3.3.whois++: UDP, length 18
11:11:36.891442 IP 4.4.4.4.whois++ > 3.3.3.3.whois++: UDP, length 18
11:11:36.891445 IP 4.4.4.4.whois++ > 3.3.3.3.whois++: UDP, length 18
```

6.10 DPDK Telemetry User Guide

The Telemetry library provides users with the ability to query DPDK for telemetry information, currently including information such as ethdev stats, ethdev port list, and eal parameters.

Note: This library is experimental and the output format may change in the future.

6.10.1 Telemetry Interface

The *Telemetry Library* opens a socket with path *<runtime_directory>/dpdk_telemetry.<version>*. The version represents the telemetry version, the latest is v2. For example, a client would connect to a socket with path */var/run/dpdk/*/dpdk_telemetry.v2* (when the primary process is run by a root user).

6.10.2 Telemetry Initialization

The library is enabled by default, however an EAL flag to enable the library exists, to provide backward compatibility for the previous telemetry library interface.

```
--telemetry
```

A flag exists to disable Telemetry also.

```
--no-telemetry
```

6.10.3 Running Telemetry

The following steps show how to run an application with telemetry support, and query information using the telemetry client python script.

1. Launch testpmd as the primary application with telemetry.

```
./app/dpdk-testpmd
```

2. Launch the telemetry client script.

```
python usertools/dpdk-telemetry.py
```

3. When connected, the script displays the following, waiting for user input.

```
Connecting to /var/run/dpdk/rte/dpdk_telemetry.v2
{"version": "DPDK 20.05.0-rc0", "pid": 60285, "max_output_len": 16384}
-->
```

4. The user can now input commands to send across the socket, and receive the response.

```
--> /
{"/": ["/", "/eal/app_params", "/eal/params", "/ethdev/list",
"/ethdev/link_status", "/ethdev/xstats", "/help", "/info"]}
--> /ethdev/list
{"/ethdev/list": [0, 1]}
```

6.11 Debug & Troubleshoot guide

DPDK applications can be designed to have simple or complex pipeline processing stages making use of single or multiple threads. Applications can use poll mode hardware devices which helps in offloading CPU cycles too. It is common to find solutions designed with

- single or multiple primary processes
- single primary and single secondary
- single primary and multiple secondaries

In all the above cases, it is tedious to isolate, debug, and understand various behaviors which occur randomly or periodically. The goal of the guide is to consolidate a few commonly seen issues for reference. Then, isolate to identify the root cause through step by step debug at various stages.

Note: It is difficult to cover all possible issues; in a single attempt. With feedback and suggestions from the community, more cases can be covered.

6.11.1 Application Overview

By making use of the application model as a reference, we can discuss multiple causes of issues in the guide. Let us assume the sample makes use of a single primary process, with various processing stages running on multiple cores. The application may also make uses of Poll Mode Driver, and libraries like service cores, mempool, mbuf, eventdev, cryptodev, QoS, and ethdev.

The overview of an application modeled using PMD is shown in Fig. 6.8.

Fig. 6.8: Overview of pipeline stage of an application

6.11.2 Bottleneck Analysis

A couple of factors that lead the design decision could be the platform, scale factor, and target. This distinct preference leads to multiple combinations, that are built using PMD and libraries of DPDK. While the compiler, library mode, and optimization flags are the components are to be constant, that affects the application too.

Is there mismatch in packet (received < desired) rate?

RX Port and associated core Fig. 6.9.

Fig. 6.9: RX packet rate compared against received rate.

1. Is the configuration for the RX setup correctly?
   - Identify if port Speed and Duplex is matching to desired values with `rte_eth_link_get`.
   - Check `DEV_RX_OFFLOAD_JUMBO_FRAME` is set with `rte_eth_dev_info_get`.
   - Check promiscuous mode if the drops do not occur for unique MAC address with `rte_eth_promiscuous_get`.
2. Is the drop isolated to certain NIC only?
   - Make use of `rte_eth_dev_stats` to identify the drops cause.
   - If there are mbuf drops, check nb_desc for RX descriptor as it might not be sufficient for the application.
   - If `rte_eth_dev_stats` shows drops are on specific RX queues, ensure RX lcore threads has enough cycles for `rte_eth_rx_burst` on the port queue pair.
   - If there are redirect to a specific port queue pair with, ensure RX lcore threads gets enough cycles.
   - Check the RSS configuration `rte_eth_dev_rss_hash_conf_get` if the spread is not even and causing drops.
   - If PMD stats are not updating, then there might be offload or configuration which is dropping the incoming traffic.
3. Is there drops still seen?
   - If there are multiple port queue pair, it might be the RX thread, RX distributor, or event RX adapter not having enough cycles.
   - If there are drops seen for RX adapter or RX distributor, try using `rte_prefetch_non_temporal` which intimates the core that the mbuf in the cache is temporary.

Is there packet drops at receive or transmit?

RX-TX port and associated cores Fig. 6.10.

Fig. 6.10: RX-TX drops

1. At RX
   - Identify if there are multiple RX queue configured for port by `nb_rx_queues` using `rte_eth_dev_info_get`.
   - Using `rte_eth_dev_stats` fetch drops in q_errors, check if RX thread is configured to fetch packets from the port queue pair.
   - Using `rte_eth_dev_stats` shows drops in `rx_nombuf`, check if RX thread has enough cycles to consume the packets from the queue.
2. At TX
   - If the TX rate is falling behind the application fill rate, identify if there are enough descriptors with `rte_eth_dev_info_get` for TX.
   - Check the `nb_pkt` in `rte_eth_tx_burst` is done for multiple packets.
   - Check `rte_eth_tx_burst` invokes the vector function call for the PMD.
   - If oerrors are getting incremented, TX packet validations are failing. Check if there queue specific offload failures.
   - If the drops occur for large size packets, check MTU and multi-segment support configured for NIC.

Is there object drops in producer point for the ring library?

Producer point for ring Fig. 6.11.

Fig. 6.11: Producer point for Rings

1. Performance issue isolation at producer
   - Use `rte_ring_dump` to validate for all single producer flag is set to `RING_F_SP_ENQ`.
   - There should be sufficient `rte_ring_free_count` at any point in time.
   - Extreme stalls in dequeue stage of the pipeline will cause `rte_ring_full` to be true.

Is there object drops in consumer point for the ring library?

Consumer point for ring Fig. 6.12.

Fig. 6.12: Consumer point for Rings

1. Performance issue isolation at consumer
   - Use `rte_ring_dump` to validate for all single consumer flag is set to `RING_F_SC_DEQ`.
   - If the desired burst dequeue falls behind the actual dequeue, the enqueue stage is not filling up the ring as required.
   - Extreme stall in the enqueue will lead to `rte_ring_empty` to be true.

Is there a variance in packet or object processing rate in the pipeline?

Memory objects close to NUMA Fig. 6.13.

Fig. 6.13: Memory objects have to be close to the device per NUMA.

1. Stall in processing pipeline can be attributes of MBUF release delays. These can be narrowed down to
   - Heavy processing cycles at single or multiple processing stages.
   - Cache is spread due to the increased stages in the pipeline.
   - CPU thread responsible for TX is not able to keep up with the burst of traffic.
   - Extra cycles to linearize multi-segment buffer and software offload like checksum, TSO, and VLAN strip.
   - Packet buffer copy in fast path also results in stalls in MBUF release if not done selectively.
   - Application logic sets `rte_pktmbuf_refcnt_set` to higher than the desired value and frequently uses `rte_pktmbuf_prefree_seg` and does not release MBUF back to mempool.
2. Lower performance between the pipeline processing stages can be
   - The NUMA instance for packets or objects from NIC, mempool, and ring should be the same.
   - Drops on a specific socket are due to insufficient objects in the pool. Use `rte_mempool_get_count` or `rte_mempool_avail_count` to monitor when drops occurs.
   - Try prefetching the content in processing pipeline logic to minimize the stalls.
3. Performance issue can be due to special cases
   - Check if MBUF continuous with `rte_pktmbuf_is_contiguous` as certain offload requires the same.
   - Use `rte_mempool_cache_create` for user threads require access to mempool objects.
   - If the variance is absent for larger huge pages, then try rte_mem_lock_page on the objects, packets, lookup tables to isolate the issue.

Is there a variance in cryptodev performance?

Crypto device and PMD Fig. 6.14.

Fig. 6.14: CRYPTO and interaction with PMD device.

1. Performance issue isolation for enqueue
   - Ensure cryptodev, resources and enqueue is running on NUMA cores.
   - Isolate if the cause of errors for err_count using `rte_cryptodev_stats`.
   - Parallelize enqueue thread for varied multiple queue pair.
2. Performance issue isolation for dequeue
   - Ensure cryptodev, resources and dequeue are running on NUMA cores.
   - Isolate if the cause of errors for err_count using `rte_cryptodev_stats`.
   - Parallelize dequeue thread for varied multiple queue pair.
3. Performance issue isolation for crypto operation
   - If the cryptodev software-assist is in use, ensure the library is built with right (SIMD) flags or check if the queue pair using CPU ISA for feature_flags AVX|SSE|NEON using `rte_cryptodev_info_get`.
   - If the cryptodev hardware-assist is in use, ensure both firmware and drivers are up to date.
4. Configuration issue isolation
   - Identify cryptodev instances with `rte_cryptodev_count` and `rte_cryptodev_info_get`.

Is user functions performance is not as expected?

Custom worker function Fig. 6.15.

Fig. 6.15: Custom worker function performance drops.

1. Performance issue isolation
   - The functions running on CPU cores without context switches are the performing scenarios. Identify lcore with `rte_lcore` and lcore index mapping with CPU using `rte_lcore_index`.
   - Use `rte_thread_get_affinity` to isolate functions running on the same CPU core.
2. Configuration issue isolation
   - Identify core role using `rte_eal_lcore_role` to identify RTE, OFF and SERVICE. Check performance functions are mapped to run on the cores.
   - For high-performance execution logic ensure running it on correct NUMA and non-master core.
   - Analyze run logic with `rte_dump_stack`, `rte_dump_registers` and `rte_memdump` for more insights.

- Make use of objdump to ensure opcode is matching to the desired state.

Is the execution cycles for dynamic service functions are not frequent?

service functions on service cores Fig. 6.16.

Fig. 6.16: functions running on service cores

1. Performance issue isolation
   - Services configured for parallel execution should have `rte_service_lcore_count` should be equal to `rte_service_lcore_count_services`.
   - A service to run parallel on all cores should return `RTE_SERVICE_CAP_MT_SAFE` for `rte_service_probe_capability` and `rte_service_map_lcore_get` returns unique lcore.
   - If service function execution cycles for dynamic service functions are not frequent?
   - If services share the lcore, overall execution should fit budget.
2. Configuration issue isolation
   - Check if service is running with `rte_service_runstate_get`.
   - Generic debug via `rte_service_dump`.

Is there a bottleneck in the performance of eventdev?

1. Check for generic configuration
   - Ensure the event devices created are right NUMA using `rte_event_dev_count` and `rte_event_dev_socket_id`.
   - Check for event stages if the events are looped back into the same queue.
   - If the failure is on the enqueue stage for events, check if queue depth with `rte_event_dev_info_get`.
2. If there are performance drops in the enqueue stage
   - Use `rte_event_dev_dump` to dump the eventdev information.
   - Periodically checks stats for queue and port to identify the starvation.
   - Check the in-flight events for the desired queue for enqueue and dequeue.

Is there a variance in traffic manager?

Traffic Manager on TX interface Fig. 6.17.

Fig. 6.17: Traffic Manager just before TX.

1. Identify the cause for a variance from expected behavior, is due to insufficient CPU cycles. Use `rte_tm_capabilities_get` to fetch features for hierarchies, WRED and priority schedulers to be offloaded hardware.
2. Undesired flow drops can be narrowed down to WRED, priority, and rates limiters.
3. Isolate the flow in which the undesired drops occur. Use `rte_tn_get_number_of_leaf_node` and flow table to ping down the leaf where drops occur.
4. Check the stats using `rte_tm_stats_update` and `rte_tm_node_stats_read` for drops for hierarchy, schedulers and WRED configurations.

Is the packet in the unexpected format?

Packet capture before and after processing Fig. 6.18.

Fig. 6.18: Capture points of Traffic at RX-TX.

1. To isolate the possible packet corruption in the processing pipeline, carefully staged capture packets are to be implemented.
   - First, isolate at NIC entry and exit.

     Use pdump in primary to allow secondary to access port-queue pair. The packets get copied over in RX|TX callback by the secondary process using ring buffers.
   - Second, isolate at pipeline entry and exit.

     Using hooks or callbacks capture the packet middle of the pipeline stage to copy the packets, which can be shared to the secondary debug process via user-defined custom rings.

Note: Use similar analysis to objects and metadata corruption.

Does the issue still persist?

The issue can be further narrowed down to the following causes.

1. If there are vendor or application specific metadata, check for errors due to META data error flags. Dumping private meta-data in the objects can give insight into details for debugging.
2. If there are multi-process for either data or configuration, check for possible errors in the secondary process where the configuration fails and possible data corruption in the data plane.
3. Random drops in the RX or TX when opening other application is an indication of the effect of a noisy neighbor. Try using the cache allocation technique to minimize the effect between applications.

6.11.3 How to develop a custom code to debug?

1. For an application that runs as the primary process only, debug functionality is added in the same process. These can be invoked by timer call-back, service core and signal handler.
2. For the application that runs as multiple processes. debug functionality in a standalone secondary process.

6.12 Enable DPDK on OpenWrt

This document describes how to enable Data Plane Development Kit (DPDK) on OpenWrt in both a virtual and physical x86 environment.

6.12.1 Introduction

The OpenWrt project is a well-known source-based router OS which provides a fully writable filesystem with package management.

6.12.2 Build OpenWrt

You can obtain OpenWrt image through https://downloads.openwrt.org/releases. To fully customize your own OpenWrt, it is highly recommended to build it from the source code. You can clone the OpenWrt source code as follows:

```
git clone https://git.openwrt.org/openwrt/openwrt.git
```

OpenWrt configuration

- Select `x86` in `Target System`
- Select `x86_64` in `Subtarget`
- Select `Build the OpenWrt SDK` for cross-compilation environment
- Select `Use glibc` in `Advanced configuration options (for developers)` then `ToolChain Options` and `C Library implementation`

Kernel configuration

The following configurations should be enabled:

- `CONFIG_VFIO_IOMMU_TYPE1=y`
- `CONFIG_VFIO_VIRQFD=y`
- `CONFIG_VFIO=y`
- `CONFIG_VFIO_NOIOMMU=y`
- `CONFIG_VFIO_PCI=y`
- `CONFIG_VFIO_PCI_MMAP=y`

- CONFIG_HUGETLBFS=y
- CONFIG_HUGETLB_PAGE=y
- CONFIG_PROC_PAGE_MONITOR=y

Build steps

For detailed OpenWrt build steps and prerequisites, please refer to the OpenWrt build guide.

After the build is completed, you can find the images and SDK in <OpenWrt Root>/bin/targets/x86/64-glibc/.

6.12.3 DPDK Cross Compilation for OpenWrt

Pre-requisites

NUMA is required to run DPDK in x86.

Note: For compiling the NUMA lib, run `libtool --version` to ensure the libtool version >= 2.2, otherwise the compilation will fail with errors.

```
git clone https://github.com/numactl/numactl.git
cd numactl
git checkout v2.0.13 -b v2.0.13
./autogen.sh
autoconf -i
export PATH=<OpenWrt SDK>/glibc/openwrt-sdk-x86-64_gcc-8.3.0_glibc.Linux-x86_64/staging_dir/
↪toolchain-x86_64_gcc-8.3.0_glibc/bin/:$PATH
./configure CC=x86_64-openwrt-linux-gnu-gcc --prefix=<OpenWrt SDK toolchain dir>
make install
```

The numa header files and lib file is generated in the include and lib folder respectively under <OpenWrt SDK toolchain dir>.

Build DPDK

To cross compile with meson build, you need to write a customized cross file first.

```
[binaries]
c = 'x86_64-openwrt-linux-gcc'
cpp = 'x86_64-openwrt-linux-cpp'
ar = 'x86_64-openwrt-linux-ar'
strip = 'x86_64-openwrt-linux-strip'

meson builddir --cross-file openwrt-cross
ninja -C builddir
```

Note: For compiling the igb_uio with the kernel version used in target machine, you need to explicitly specify kernel_dir in meson_options.txt.

6.12.4 Running DPDK application on OpenWrt

Virtual machine

- Extract the boot image

```
gzip -d openwrt-x86-64-combined-ext4.img.gz
```

- Launch Qemu

```
qemu-system-x86_64 \
        -cpu host \
        -smp 8 \
        -enable-kvm \
        -M q35 \
        -m 2048M \
        -object memory-backend-file,id=mem,size=2048M,mem-path=/tmp/hugepages,share=on \
        -drive file=<Your OpenWrt images folder>/openwrt-x86-64-combined-ext4.img,id=d0,
↪if=none,bus=0,unit=0 \
        -device ide-hd,drive=d0,bus=ide.0 \
        -net nic,vlan=0 \
        -net nic,vlan=1 \
        -net user,vlan=1 \
        -display none \
```

Physical machine

You can use the dd tool to write the OpenWrt image to the drive you want to write the image on.

```
dd if=openwrt-18.06.1-x86-64-combined-squashfs.img of=/dev/sdX
```

Where sdX is name of the drive. (You can find it though `fdisk -l`)

Running DPDK

More detailed info about how to run a DPDK application please refer to `Running DPDK Applications` section of *the DPDK documentation*.

Note: You need to install pre-built NUMA libraries (including soft link) to /usr/lib64 in OpenWrt.

CHAPTER

SEVEN

DPDK TOOLS USER GUIDES

7.1 dpdk-procinfo Application

The dpdk-procinfo application is a Data Plane Development Kit (DPDK) application that runs as a DPDK secondary process and is capable of retrieving port statistics, resetting port statistics, printing DPDK memory information and displaying debug information for port. This application extends the original functionality that was supported by dump_cfg.

7.1.1 Running the Application

The application has a number of command line options:

```
./$(RTE_TARGET)/app/dpdk-procinfo -- -m | [-p PORTMASK] [--stats | --xstats |
--stats-reset | --xstats-reset] [ --show-port | --show-tm | --show-crypto |
--show-ring[=name] | --show-mempool[=name] | --iter-mempool=name ]
```

Parameters

-p PORTMASK: Hexadecimal bitmask of ports to configure.

–stats The stats parameter controls the printing of generic port statistics. If no port mask is specified stats are printed for all DPDK ports.

–xstats The xstats parameter controls the printing of extended port statistics. If no port mask is specified xstats are printed for all DPDK ports.

–stats-reset The stats-reset parameter controls the resetting of generic port statistics. If no port mask is specified, the generic stats are reset for all DPDK ports.

–xstats-reset The xstats-reset parameter controls the resetting of extended port statistics. If no port mask is specified xstats are reset for all DPDK ports.

-m: Print DPDK memory information.

–show-port The show-port parameter displays port level various configuration information associated to RX port queue pair.

–show-tm The show-tm parameter displays per port traffic manager settings, current configurations and statistics.

–show-crypto The show-crypto parameter displays available cryptodev configurations, settings and stats per node.

–show-ring[=name] The show-ring parameter display current allocation of all ring with debug information. Specifying the name allows to display details for specific ring. For invalid or no ring name, whole list is dump.

–show-mempool[=name] The show-mempool parameter display current allocation of all mempool debug information. Specifying the name allows to display details for specific mempool. For invalid or no mempool name, whole list is dump.

–iter-mempool=name The iter-mempool parameter iterates and displays mempool elements specified by name. For invalid or no mempool name no elements are displayed.

7.1.2 Limitations

- dpdk-procinfo should run alongside primary process with same DPDK version.
- When running `dpdk-procinfo` with shared library mode, it is required to pass the same NIC PMD libraries as used for the primary application. Any mismatch in PMD library arguments can lead to undefined behavior and results affecting primary application too.
- Stats retrieval using `dpdk-procinfo` is not supported for virtual devices like PCAP and TAP.
- Since default DPDK EAL arguments for `dpdk-procinfo` are `-c1, -n4 & --proc-type=secondary`, It is not expected that the user passes any EAL arguments.

7.2 dpdk-pdump Application

The `dpdk-pdump` tool is a Data Plane Development Kit (DPDK) tool that runs as a DPDK secondary process and is capable of enabling packet capture on dpdk ports.

Note:

- The `dpdk-pdump` tool can only be used in conjunction with a primary application which has the packet capture framework initialized already. In dpdk, only the `testpmd` is modified to initialize packet capture framework, other applications remain untouched. So, if the `dpdk-pdump` tool has to be used with any application other than the testpmd, user needs to explicitly modify that application to call packet capture framework initialization code. Refer `app/test-pmd/testpmd.c` code to see how this is done.
- The `dpdk-pdump` tool depends on libpcap based PMD which is disabled by default in the build configuration files, owing to an external dependency on the libpcap development files which must be installed on the board. Once the libpcap development files are installed, the libpcap based PMD can be enabled by setting CONFIG_RTE_LIBRTE_PMD_PCAP=y and recompiling the DPDK.
- The `dpdk-pdump` tool runs as a DPDK secondary process. It exits when the primary application exits.

7.2.1 Running the Application

The tool has a number of command line options:

```
./build/app/dpdk-pdump --
                       [--multi]
                       --pdump '(port=<port id> | device_id=<pci id or vdev name>),
                                (queue=<queue_id>),
                                (rx-dev=<iface or pcap file> |
                                 tx-dev=<iface or pcap file>),
                                [ring-size=<ring size>],
                                [mbuf-size=<mbuf data size>],
                                [total-num-mbufs=<number of mbufs>]'
```

The `--multi` command line option is optional argument. If passed, capture will be running on unique cores for all `--pdump` options. If ignored, capture will be running on single core for all `--pdump` options.

The `--pdump` command line option is mandatory and it takes various sub arguments which are described in below section.

Note:

- Parameters inside the parentheses represents mandatory parameters.
- Parameters inside the square brackets represents optional parameters.
- Multiple instances of `--pdump` can be passed to capture packets on different port and queue combinations.

The `--pdump` parameters

`port`: Port id of the eth device on which packets should be captured.

`device_id`: PCI address (or) name of the eth device on which packets should be captured.

Note:

- As of now the `dpdk-pdump` tool cannot capture the packets of virtual devices in the primary process due to a bug in the ethdev library. Due to this bug, in a multi process context, when the primary and secondary have different ports set, then the secondary process (here the `dpdk-pdump` tool) overwrites the `rte_eth_devices[]` entries of the primary process.

`queue`: Queue id of the eth device on which packets should be captured. The user can pass a queue value of `*` to enable packet capture on all queues of the eth device.

`rx-dev`: Can be either a pcap file name or any Linux iface.

`tx-dev`: Can be either a pcap file name or any Linux iface.

Note:

- To receive ingress packets only, `rx-dev` should be passed.
- To receive egress packets only, `tx-dev` should be passed.

- To receive ingress and egress packets separately `rx-dev` and `tx-dev` should both be passed with the different file names or the Linux iface names.
- To receive ingress and egress packets together, `rx-dev` and `tx-dev` should both be passed with the same file name or the same Linux iface name.

`ring-size`: Size of the ring. This value is used internally for ring creation. The ring will be used to enqueue the packets from the primary application to the secondary. This is an optional parameter with default size 16384.

`mbuf-size`: Size of the mbuf data. This is used internally for mempool creation. Ideally this value must be same as the primary application's mempool's mbuf data size which is used for packet RX. This is an optional parameter with default size 2176.

`total-num-mbufs`: Total number mbufs in mempool. This is used internally for mempool creation. This is an optional parameter with default value 65535.

7.2.2 Example

```
$ sudo ./build/app/dpdk-pdump -l 3 -- --pdump 'port=0,queue=*,rx-dev=/tmp/rx.pcap'
$ sudo ./build/app/dpdk-pdump -l 3,4,5 -- --multi --pdump 'port=0,queue=*,rx-dev=/tmp/rx-1.pcap
↪' --pdump 'port=1,queue=*,rx-dev=/tmp/rx-2.pcap'
```

7.3 dpdk-pmdinfo Application

The `dpdk-pmdinfo` tool is a Data Plane Development Kit (DPDK) utility that can dump a PMDs hardware support info.

7.3.1 Running the Application

The tool has a number of command line options:

```
dpdk-pmdinfo [-hrtp] [-d <pci id file] <elf-file>

-h, --help            Show a short help message and exit
-r, --raw             Dump as raw json strings
-d FILE, --pcidb=FILE Specify a pci database to get vendor names from
-t, --table           Output information on hw support as a hex table
-p, --plugindir       Scan dpdk for autoload plugins
```

Note:

- Parameters inside the square brackets represents optional parameters.

7.4 dpdk-devbind Application

The dpdk-devbind tool is a Data Plane Development Kit (DPDK) utility that helps binding and unbinding devices from specific drivers. As well as checking their status in that regard.

7.4.1 Running the Application

The tool has a number of command line options:

```
dpdk-devbind [options] DEVICE1 DEVICE2 ....
```

7.4.2 OPTIONS

- `--help, --usage`

  Display usage information and quit

- `-s, --status`

  Print the current status of all known network interfaces. For each device, it displays the PCI domain, bus, slot and function, along with a text description of the device. Depending upon whether the device is being used by a kernel driver, the `igb_uio` driver, or no driver, other relevant information will be displayed: - the Linux interface name e.g. `if=eth0` - the driver being used e.g. `drv=igb_uio` - any suitable drivers not currently using that device e.g. `unused=igb_uio` NOTE: if this flag is passed along with a bind/unbind option, the status display will always occur after the other operations have taken place.

- `-b driver, --bind=driver`

  Select the driver to use or "none" to unbind the device

- `-u, --unbind`

  Unbind a device (Equivalent to `-b none`)

- `--force`

  By default, devices which are used by Linux - as indicated by having routes in the routing table - cannot be modified. Using the `--force` flag overrides this behavior, allowing active links to be forcibly unbound. WARNING: This can lead to loss of network connection and should be used with caution.

Warning: Due to the way VFIO works, there are certain limitations to which devices can be used with VFIO. Mainly it comes down to how IOMMU groups work. Any Virtual Function device can be used with VFIO on its own, but physical devices will require either all ports bound to VFIO, or some of them bound to VFIO while others not being bound to anything at all.

If your device is behind a PCI-to-PCI bridge, the bridge will then be part of the IOMMU group in which your device is in. Therefore, the bridge driver should also be unbound from the bridge PCI device for VFIO to work with devices behind the bridge.

Warning: While any user can run the `dpdk-devbind.py` script to view the status of the network ports, binding or unbinding network ports requires root privileges.

7.4.3 Examples

To display current device status:

```
dpdk-devbind --status
```

To bind eth1 from the current driver and move to use igb_uio:

```
dpdk-devbind --bind=igb_uio eth1
```

To unbind 0000:01:00.0 from using any driver:

```
dpdk-devbind -u 0000:01:00.0
```

To bind 0000:02:00.0 and 0000:02:00.1 to the ixgbe kernel driver:

```
dpdk-devbind -b ixgbe 02:00.0 02:00.1
```

To check status of all network ports, assign one to the igb_uio driver and check status again:

```
# Check the status of the available devices.
dpdk-devbind --status
Network devices using DPDK-compatible driver
============================================
<none>

Network devices using kernel driver
===================================
0000:0a:00.0 '82599ES 10-Gigabit' if=eth2 drv=ixgbe unused=


# Bind the device to igb_uio.
sudo dpdk-devbind -b igb_uio 0000:0a:00.0


# Recheck the status of the devices.
dpdk-devbind --status
Network devices using DPDK-compatible driver
============================================
0000:0a:00.0 '82599ES 10-Gigabit' drv=igb_uio unused=
```

7.5 dpdk-test-bbdev Application

The `dpdk-test-bbdev` tool is a Data Plane Development Kit (DPDK) utility that allows measuring performance parameters of PMDs available in the bbdev framework. Available tests available for execution are: latency, throughput, validation, bler and sanity tests. Execution of tests can be customized using various parameters passed to a python running script.

7.5.1 Compiling the Application

Step 1: PMD setting

The `dpdk-test-bbdev` tool depends on crypto device drivers PMD which are disabled by default in the build configuration file `common_base`. The bbdevice drivers PMD which should be tested can be enabled by setting

```
CONFIG_RTE_LIBRTE_PMD_<name>=y
```

Setting example for (*baseband_turbo_sw*) PMD

```
CONFIG_RTE_LIBRTE_PMD_BBDEV_TURBO_SW=y
```

Step 2: Build the application

Execute the `dpdk-setup.sh` script to build the DPDK library together with the `dpdk-test-bbdev` application.

Initially, the user must select a DPDK target to choose the correct target type and compiler options to use when building the libraries. The user must have all libraries, modules, updates and compilers installed in the system prior to this, as described in the earlier chapters in this Getting Started Guide.

7.5.2 Running the Application

The tool application has a number of command line options:

```
python test-bbdev.py [-h] [-p TESTAPP_PATH] [-e EAL_PARAMS] [-t TIMEOUT]
                     [-c TEST_CASE [TEST_CASE ...]]
                     [-v TEST_VECTOR [TEST_VECTOR...]] [-n NUM_OPS]
                     [-b BURST_SIZE [BURST_SIZE ...]] [-l NUM_LCORES]
                     [-t MAX_ITERS [MAX_ITERS ...]]
                     [-s SNR [SNR ...]]
```

command-line Options

The following are the command-line options:

-h, --help
Shows help message and exit.

-p TESTAPP_PATH, --testapp_path TESTAPP_PATH
Indicates the path to the bbdev test app. If not specified path is set based on *$RTE_SDK* environment variable concatenated with "*/build/app/testbbdev*".

-e EAL_PARAMS, --eal_params EAL_PARAMS
Specifies EAL arguments which are passed to the test app. For more details, refer to DPDK documentation at *EAL parameters*.

-t TIMEOUT, --timeout TIMEOUT
Specifies timeout in seconds. If not specified timeout is set to 300 seconds.

-c TEST_CASE [TEST_CASE ...], --test_cases TEST_CASE [TEST_CASE ...]
Defines test cases to run. If not specified all available tests are run.

Example usage:

`./test-bbdev.py -c validation`
Runs validation test suite

`./test-bbdev.py -c latency throughput`
Runs latency and throughput test suites

`-v TEST_VECTOR [TEST_VECTOR ...], --test_vector TEST_VECTOR [TEST_VECTOR ...]`
Specifies paths to the test vector files. If not specified path is set based on *$RTE_SDK* environment variable concatenated with "*/app/test-bbdev/test_vectors/bbdev_null.data*" and indicates default data file.

Example usage:

`./test-bbdev.py -v app/test-bbdev/test_vectors/turbo_dec_test1.data`
Fills vector based on turbo_dec_test1.data file and runs all tests

`./test-bbdev.py -v turbo_dec_test1.data turbo_enc_test2.data`
The bbdev test app is executed twice. First time vector is filled based on *turbo_dec_test1.data* file and second time based on *turb_enc_test2.data* file. For both executions all tests are run.

`-n NUM_OPS, --num_ops NUM_OPS`
Specifies number of operations to process on device. If not specified num_ops is set to 32 operations.

`-l NUM_LCORES, --num_lcores NUM_LCORES`
Specifies number of lcores to run. If not specified num_lcores is set according to value from RTE configuration (EAL coremask)

`-b BURST_SIZE [BURST_SIZE ...], --burst-size BURST_SIZE [BURST_SIZE ...]`
Specifies operations enqueue/dequeue burst size. If not specified burst_size is set to 32. Maximum is 512.

`-t MAX_ITERS [MAX_ITERS ...], --iter_max MAX_ITERS [MAX_ITERS ...]`
Specifies LDPC decoder operations maximum number of iterations for throughput and bler tests. If not specified iter_max is set to 6.

`-s SNR [SNR ...], --snr SNR [SNR ...]`
Specifies for LDPC decoder operations the SNR in dB used when generating LLRs for bler tests. If not specified snr is set to 0 dB.

Test Cases

There are 7 main test cases that can be executed using testbbdev tool:

- **Sanity checks [-c unittest]**
  - Performs sanity checks on BBDEV interface, validating basic functionality
- **Validation tests [-c validation]**
  - Performs full operation of enqueue and dequeue
  - Compares the dequeued data buffer with a expected values in the test vector (TV) being used
  - Fails if any dequeued value does not match the data in the TV
- **Offload Cost measurement [-c offload]**

  - Measures the CPU cycles consumed from the receipt of a user enqueue until it is put on the device queue
  - **The test measures 4 metrics**
    (a) *SW Enq Offload Cost*: Software only enqueue offload cost, the cycle counts and time (us) from the point the enqueue API is called until the point the operation is put on the accelerator queue.
    (b) *Acc Enq Offload Cost*: The cycle count and time (us) from the point the operation is put on the accelerator queue until the return from enqueue.
    (c) *SW Deq Offload Cost*: Software dequeue cost, the cycle counts and time (us) consumed to dequeue one operation.
    (d) *Empty Queue Enq Offload Cost*: The cycle count and time (us) consumed to dequeue from an empty queue.

- **Latency measurement [-c latency]**
  - Measures the time consumed from the first enqueue until the first appearance of a dequeued result
  - This measurement represents the full latency of a bbdev operation (encode or decode) to execute

- **Poll-mode Throughput measurement [-c throughput]**
  - Performs full operation of enqueue and dequeue
  - Executes in poll mode
  - Measures the achieved throughput on a subset or all available CPU cores
  - Dequeued data is not validated against expected values stored in TV
  - Results are printed in million operations per second and million bits per second

- **BLER measurement [-c bler]**
  - Performs full operation of enqueue and dequeue
  - Measures the achieved throughput on a subset or all available CPU cores
  - Computed BLER (Block Error Rate, ratio of blocks not decoded at a given SNR) in % based on the total number of operations.

- **Interrupt-mode Throughput [-c interrupt]**
  - Similar to Throughput test case, but using interrupts. No polling.

Parameter Globbing

Thanks to the globbing functionality in python test-bbdev.py script allows to run tests with different set of vector files without giving all of them explicitly.

Example usage for 4G:

```
./test-bbdev.py -v app/test-bbdev/test_vectors/turbo_<enc/dec>_c<c>_k<k>_r<r>_e<e>_<extra-info>
↪.data
```

It runs all tests with following vectors:

- bbdev_null.data
- turbo_dec_c1_k6144_r0_e34560_sbd_negllr.data
- turbo_enc_c1_k40_r0_e1196_rm.data
- turbo_enc_c2_k5952_r0_e17868_crc24b.data
- turbo_dec_c1_k40_r0_e17280_sbd_negllr.data
- turbo_dec_c1_k6144_r0_e34560_sbd_posllr.data
- turbo_enc_c1_k40_r0_e272_rm.data
- turbo_enc_c3_k4800_r2_e14412_crc24b.data
- turbo_dec_c1_k6144_r0_e10376_crc24b_sbd_negllr_high_snr.data
- turbo_dec_c2_k3136_r0_e4920_sbd_negllr_crc24b.data
- turbo_enc_c1_k6144_r0_e120_rm_rvidx.data
- turbo_enc_c4_k4800_r2_e14412_crc24b.data
- turbo_dec_c1_k6144_r0_e10376_crc24b_sbd_negllr_low_snr.data
- turbo_dec_c2_k3136_r0_e4920_sbd_negllr.data
- turbo_enc_c1_k6144_r0_e18444.data
- turbo_dec_c1_k6144_r0_e34560_negllr.data
- turbo_enc_c1_k40_r0_e1190_rm.data
- turbo_enc_c1_k6144_r0_e18448_crc24a.data
- turbo_dec_c1_k6144_r0_e34560_posllr.data
- turbo_enc_c1_k40_r0_e1194_rm.data
- turbo_enc_c1_k6144_r0_e32256_crc24b_rm.data

```
./test-bbdev.py -v app/test-bbdev/turbo_*_default.data
```

It runs all tests with "default" vectors.

- turbo_dec_default.data is a soft link to turbo_dec_c1_k6144_r0_e10376_crc24b_sbd_negllr_high_s
  data
- turbo_enc_default.data is a soft link to turbo_enc_c1_k6144_r0_e32256_crc24b_rm.
  data
- ldpc_dec_default.data is a soft link to ldpc_dec_v6563.data
- ldpc_enc_default.data is a soft link to ldpc_enc_c1_k8148_r0_e9372_rm.data

7.5.3 Running Tests

All default reference test-vectors are stored in the test_vector directory below. The prefix trivially defines which type of operation is included : turbo_enc, turbo_dec, ldpc_enc, ldpc_dec. The details of the configuration are captured in the file but some vector name refer more explicitly processing specificity such as 'HARQ' when HARQ retransmission is used, 'loopback' when the data is purely read/written for external DDR, lbrm when limited buffer rate matching is expected, or crc_fail when a CRC failure is expected. They are chosen to have a good coverage across sizes and processing parameters while still keeping their number limited as part of sanity regression.

Shortened tree of isg_cid-wireless_dpdk_ae with dpdk compiled for x86_64-native-linux-icc target:

```
|-- app
    |-- test-bbdev
        |-- test_vectors

|-- x86_64-native-linux-icc
    |-- app
        |-- testbbdev
```

All bbdev devices

```
./test-bbdev.py -p ../../x86_64-native-linux-icc/app/testbbdev
-v turbo_dec_default.data
```

It runs all available tests using the test vector filled based on *turbo_dec_default.data* file. By default number of operations to process on device is set to 32, timeout is set to 300s and operations enqueue/dequeue burst size is set to 32. Moreover a bbdev (*baseband_null*) device will be created.

baseband turbo_sw device

```
./test-bbdev.py -p ../../x86_64-native-linux-icc/app/testbbdev
-e="--vdev=baseband_turbo_sw" -t 120 -c validation
-v ./test_vectors/* -n 64 -b 8 32
```

It runs **validation** test for each vector file that matches the given pattern. Number of operations to process on device is set to 64 and operations timeout is set to 120s and enqueue/dequeue burst size is set to 8 and to 32. Moreover a bbdev (*baseband_turbo_sw*) device will be created.

bbdev null device

Executing bbdev null device with *bbdev_null.data* helps in measuring the overhead introduced by the bbdev framework.

```
./test-bbdev.py -e="--vdev=baseband_null0"
-v ./test_vectors/bbdev_null.data
```

Note:

baseband_null device does not have to be defined explicitly as it is created by default.

7.5.4 Test Vector files

Test Vector files contain the data which is used to set turbo decoder/encoder parameters and buffers for validation purpose. New test vector files should be stored in `app/test-bbdev/test_vectors/` directory. Detailed description of the syntax of the test vector files is in the following section.

Basic principles for test vector files

Line started with # is treated as a comment and is ignored.

If variable is a chain of values, values should be separated by a comma. If assignment is split into several lines, each line (except the last one) has to be ended with a comma. There is no comma after last value in last line. Correct assignment should look like the following:

```
variable =
value, value, value, value,
value, value
```

In case where variable is a single value correct assignment looks like the following:

```
variable =
value
```

Length of chain variable is calculated by parser. Can not be defined explicitly.

Variable op_type has to be defined as a first variable in file. It specifies what type of operations will be executed. For 4G decoder op_type has to be set to `RTE_BBDEV_OP_TURBO_DEC` and for 4G encoder to `RTE_BBDEV_OP_TURBO_ENC`.

Full details of the meaning and valid values for the below fields are documented in *rte_bbdev_op.h*

Turbo decoder test vectors template

For turbo decoder it has to be always set to `RTE_BBDEV_OP_TURBO_DEC`

```
op_type =
RTE_BBDEV_OP_TURBO_DEC
```

Chain of uint32_t values. Note that it is possible to define more than one input/output entries which will result in chaining two or more data structures for *segmented Transport Blocks*

```
input0 =
0x00000000, 0x7f817f00, 0x7f7f8100, 0x817f8100, 0x81008100, 0x7f818100, 0x81817f00, 0x7f818100,
0x81007f00, 0x7f818100, 0x817f8100, 0x81817f00, 0x81008100, 0x817f7f00, 0x7f7f8100, 0x81817f00
```

Chain of uint32_t values

```
input1 =
0x7f7f0000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000
```

Chain of uint32_t values

```
input2 =
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000
```

Chain of uint32_t values

```
hard_output0 =
0xa7d6732e
```

Chain of uint32_t values

```
hard_output1 =
0xa61
```

Chain of uint32_t values

```
soft_output0 =
0x817f817f, 0x7f817f7f, 0x81818181, 0x817f7f81, 0x7f818181, 0x8181817f, 0x817f817f, 0x8181817f
```

Chain of uint32_t values

```
soft_output1 =
0x817f7f81, 0x7f7f7f81, 0x7f7f8181
```

uint32_t value

```
e =
44
```

uint16_t value

```
k =
40
```

uint8_t value

```
rv_index =
0
```

uint8_t value

```
iter_max =
8
```

uint8_t value

```
iter_min =
4
```

uint8_t value

```
expected_iter_count =
8
```

uint8_t value

```
ext_scale =
15
```

uint8_t value

```
num_maps =
0
```

Chain of flags for LDPC decoder operation based on the rte_bbdev_op_td_flag_bitmasks:

Example:

```
op_flags =
RTE_BBDEV_TURBO_SUBBLOCK_DEINTERLEAVE, RTE_BBDEV_TURBO_EQUALIZER,
RTE_BBDEV_TURBO_SOFT_OUTPUT
```

Chain of operation statuses that are expected after operation is performed. Following statuses can be used:

- DMA
- FCW
- CRC
- OK

OK means no errors are expected. Cannot be used with other values.

```
expected_status =
FCW, CRC
```

Turbo encoder test vectors template

For turbo encoder it has to be always set to RTE_BBDEV_OP_TURBO_ENC

```
op_type =
RTE_BBDEV_OP_TURBO_ENC
```

Chain of uint32_t values

```
input0 =
0x11d2bcac, 0x4d
```

Chain of uint32_t values

```
output0 =
0xd2399179, 0x640eb999, 0x2cbaf577, 0xaf224ae2, 0x9d139927, 0xe6909b29,
0xa25b7f47, 0x2aa224ce, 0x79f2
```

uint32_t value

```
e =
272
```

uint16_t value

```
k =
40
```

uint16_t value

```
ncb =
192
```

uint8_t value

```
rv_index =
0
```

Chain of flags for LDPC decoder operation based on the rte_bbdev_op_te_flag_bitmasks:

RTE_BBDEV_TURBO_ENC_SCATTER_GATHER is used to indicate the parser to force the input data to be memory split and formed as a segmented mbuf.

```
op_flags =
RTE_BBDEV_TURBO_RATE_MATCH
```

Chain of operation statuses that are expected after operation is performed. Following statuses can be used:

- DMA
- FCW
- OK

OK means no errors are expected. Cannot be used with other values.

```
expected_status =
OK
```

LDPC decoder test vectors template

For LDPC decoder it has to be always set to RTE_BBDEV_OP_LDPC_DEC

```
op_type =
RTE_BBDEV_OP_LDPC_DEC
```

Chain of uint32_t values. Note that it is possible to define more than one input/output entries which will result in chaining two or more data structures for *segmented Transport Blocks*

```
input0 =
0x00000000, 0x7f817f00, 0x7f7f8100, 0x817f8100, 0x81008100, 0x7f818100, 0x81817f00, 0x7f818100,
0x81007f00, 0x7f818100, 0x817f8100, 0x81817f00, 0x81008100, 0x817f7f00, 0x7f7f8100, 0x81817f00
```

```
output0 =
0xa7d6732e
```

uint8_t value

```
basegraph=
1
```

uint16_t value

```
z_c=
224
```

uint16_t value

```
n_cb=
14784
```

uint8_t value

```
q_m=
1
```

uint16_t value

```
n_filler=
40
```

uint32_t value

```
e=
13072
```

uint8_t value

```
rv_index=
2
```

uint8_t value

```
code_block_mode=
1
```

uint8_t value

```
iter_max=
20
```

uint8_t value

```
expected_iter_count=
8
```

Chain of flags for LDPC decoder operation based on the rte_bbdev_op_ldpcdec_flag_bitmasks:

Example:

```
op_flags =
RTE_BBDEV_LDPC_ITERATION_STOP_ENABLE, RTE_BBDEV_LDPC_HQ_COMBINE_OUT_ENABLE,
RTE_BBDEV_LDPC_HQ_COMBINE_IN_ENABLE, RTE_BBDEV_LDPC_HARQ_6BIT_COMPRESSION
```

Chain of operation statuses that are expected after operation is performed. Following statuses can be used:

- `OK` : No error reported.
- `SYN` : LDPC syndrome parity check is failing.
- `CRC` : CRC parity check is failing when CRC check operation is included.
- `SYNCRC` : Both CRC and LDPC syndromes parity checks are failing.

`OK` means no errors are expected. Cannot be used with other values.

```
expected_status =
CRC
```

LDPC encoder test vectors template

For turbo encoder it has to be always set to RTE_BBDEV_OP_LDPC_ENC

```
op_type =
RTE_BBDEV_OP_LDPC_ENC
```

Chain of uint32_t values

```
input0 =
0x11d2bcac, 0x4d
```

Chain of uint32_t values

```
output0 =
0xd2399179, 0x640eb999, 0x2cbaf577, 0xaf224ae2, 0x9d139927, 0xe6909b29,
0xa25b7f47, 0x2aa224ce, 0x79f2
```

uint8_t value

```
basegraph=
1
```

uint16_t value

```
z_c=
52
```

uint16_t value

```
n_cb=
3432
```

uint8_t value

```
q_m=
6
```

uint16_t value

```
n_filler=
0
```

uint32_t value

```
e =
1380
```

uint8_t value

```
rv_index =
1
```

uint8_t value

```
code_block_mode =
1
```

Chain of flags for LDPC encoder operation based on the rte_bbdev_op_ldpcenc_flag_bitmasks:

```
op_flags =
RTE_BBDEV_LDPC_RATE_MATCH
```

Chain of operation statuses that are expected after operation is performed. Following statuses can be used:

- DMA
- FCW
- OK

OK means no errors are expected. Cannot be used with other values.

```
expected_status =
OK
```

7.6 dpdk-test-crypto-perf Application

The dpdk-test-crypto-perf tool is a Data Plane Development Kit (DPDK) utility that allows measuring performance parameters of PMDs available in the crypto tree. There are available two measurement types: throughput and latency. User can use multiply cores to run tests on but only one type of crypto PMD can be measured during single application execution. Cipher parameters, type of device, type of operation and chain mode have to be specified in the command line as application parameters. These parameters are checked using device capabilities structure.

7.6.1 Limitations

On hardware devices the cycle-count doesn't always represent the actual offload cost. The cycle-count only represents the offload cost when the hardware accelerator is not fully loaded, when loaded the cpu cycles freed up by the offload are still consumed by the test tool and included in the cycle-count. These cycles are consumed by retries and inefficient API calls enqueuing and dequeuing smaller bursts than specified by the cmdline parameter. This results in a larger cycle-count measurement and should not be interpreted as an offload cost measurement. Using "pmd-cyclecount" mode will give a better idea of actual costs of hardware acceleration.

On hardware devices the throughput measurement is not necessarily the maximum possible for the device, e.g. it may be necessary to use multiple cores to keep the hardware accelerator fully loaded and so measure maximum throughput.

7.6.2 Compiling the Application

Step 1: PMD setting

The `dpdk-test-crypto-perf` tool depends on crypto device drivers PMD which are disabled by default in the build configuration file `common_base`. The crypto device drivers PMD which should be tested can be enabled by setting:

```
CONFIG_RTE_LIBRTE_PMD_<name>=y
```

Setting example for open ssl PMD:

```
CONFIG_RTE_LIBRTE_PMD_OPENSSL=y
```

Step 2: Linearization setting

It is possible linearized input segmented packets just before crypto operation for devices which doesn't support scatter-gather, and allows to measure performance also for this use case.

To set on the linearization options add below definition to the `cperf_ops.h` file:

```
#define CPERF_LINEARIZATION_ENABLE
```

Step 3: Build the application

Execute the `dpdk-setup.sh` script to build the DPDK library together with the `dpdk-test-crypto-perf` application.

Initially, the user must select a DPDK target to choose the correct target type and compiler options to use when building the libraries. The user must have all libraries, modules, updates and compilers installed in the system prior to this, as described in the earlier chapters in this Getting Started Guide.

7.6.3 Running the Application

The tool application has a number of command line options:

```
dpdk-test-crypto-perf [EAL Options] -- [Application Options]
```

EAL Options

The following are the EAL command-line options that can be used in conjunction with the `dpdk-test-crypto-perf` application. See the DPDK Getting Started Guides for more information on these options.

- `-c <COREMASK>` or `-l <CORELIST>`

  Set the hexadecimal bitmask of the cores to run on. The corelist is a list cores to use.

- `-w <PCI>`

  Add a PCI device in white list.

- `--vdev <driver><id>`

  Add a virtual device.

Application Options

The following are the application command-line options:

- `--ptest type`

  Set test type, where `type` is one of the following:

  ```
  throughput
  latency
  verify
  pmd-cyclecount
  ```

- `--silent`

  Disable options dump.

- `--pool-sz <n>`

  Set the number of mbufs to be allocated in the mbuf pool.

- `--total-ops <n>`

  Set the number of total operations performed.

- `--burst-sz <n>`

  Set the number of packets per burst.

  This can be set as:

  - Single value (i.e. `--burst-sz 16`)
  - Range of values, using the following structure `min:inc:max`, where `min` is minimum size, `inc` is the increment size and `max` is the maximum size (i.e. `--burst-sz 16:2:32`)
  - List of values, up to 32 values, separated in commas (i.e. `--burst-sz 16,24,32`)

- `--buffer-sz <n>`

  Set the size of single packet (plaintext or ciphertext in it).

  This can be set as:

  - Single value (i.e. `--buffer-sz 16`)
  - Range of values, using the following structure `min:inc:max`, where `min` is minimum size, `inc` is the increment size and `max` is the maximum size (i.e. `--buffer-sz 16:2:32`)
  - List of values, up to 32 values, separated in commas (i.e. `--buffer-sz 32,64,128`)

- `--imix <n>`

  Set the distribution of packet sizes.

  A list of weights must be passed, containing the same number of items than buffer-sz, so each item in this list will be the weight of the packet size on the same position in the buffer-sz parameter (a list have to be passed in that parameter).

  Example:

To test a distribution of 20% packets of 64 bytes, 40% packets of 100 bytes and 40% packets of 256 bytes, the command line would be: `--buffer-sz 64,100,256 --imix 20,40,40`. Note that the weights do not have to be percentages, so using `--imix 1,2,2` would result in the same distribution

- `--segment-sz <n>`

  Set the size of the segment to use, for Scatter Gather List testing. By default, it is set to the size of the maximum buffer size, including the digest size, so a single segment is created.

- `--devtype <name>`

  Set device type, where `name` is one of the following:

```
crypto_null
crypto_aesni_mb
crypto_aesni_gcm
crypto_openssl
crypto_qat
crypto_snow3g
crypto_kasumi
crypto_zuc
crypto_dpaa_sec
crypto_dpaa2_sec
crypto_armv8
crypto_scheduler
crypto_mvsam
```

- `--optype <name>`

  Set operation type, where `name` is one of the following:

```
cipher-only
auth-only
cipher-then-auth
auth-then-cipher
aead
pdcp
```

  For GCM/CCM algorithms you should use aead flag.

- `--sessionless`

  Enable session-less crypto operations mode.

- `--out-of-place`

  Enable out-of-place crypto operations mode.

- `--test-file <name>`

  Set test vector file path. See the Test Vector File chapter.

- `--test-name <name>`

  Set specific test name section in the test vector file.

- `--cipher-algo <name>`

  Set cipher algorithm name, where `name` is one of the following:

```
3des-cbc
3des-ecb
3des-ctr
aes-cbc
aes-ctr
aes-ecb
aes-f8
aes-xts
arc4
null
kasumi-f8
snow3g-uea2
zuc-eea3
```

- `--cipher-op <mode>`

  Set cipher operation mode, where `mode` is one of the following:

```
encrypt
decrypt
```

- `--cipher-key-sz <n>`

  Set the size of cipher key.

- `--cipher-iv-sz <n>`

  Set the size of cipher iv.

- `--auth-algo <name>`

  Set authentication algorithm name, where `name` is one of the following:

```
3des-cbc
aes-cbc-mac
aes-cmac
aes-gmac
aes-xcbc-mac
md5
md5-hmac
sha1
sha1-hmac
sha2-224
sha2-224-hmac
sha2-256
sha2-256-hmac
sha2-384
sha2-384-hmac
sha2-512
sha2-512-hmac
kasumi-f9
snow3g-uia2
zuc-eia3
```

- `--auth-op <mode>`

  Set authentication operation mode, where `mode` is one of the following:

```
verify
generate
```

- `--auth-key-sz <n>`

Set the size of authentication key.

- `--auth-iv-sz <n>`

  Set the size of auth iv.

- `--aead-algo <name>`

  Set AEAD algorithm name, where `name` is one of the following:

  ```
  aes-ccm
  aes-gcm
  ```

- `--aead-op <mode>`

  Set AEAD operation mode, where `mode` is one of the following:

  ```
  encrypt
  decrypt
  ```

- `--aead-key-sz <n>`

  Set the size of AEAD key.

- `--aead-iv-sz <n>`

  Set the size of AEAD iv.

- `--aead-aad-sz <n>`

  Set the size of AEAD aad.

- `--digest-sz <n>`

  Set the size of digest.

- `--desc-nb <n>`

  Set number of descriptors for each crypto device.

- `--pmd-cyclecount-delay-ms <n>`

  Add a delay (in milliseconds) between enqueue and dequeue in pmd-cyclecount benchmarking mode (useful when benchmarking hardware acceleration).

- `--csv-friendly`

  Enable test result output CSV friendly rather than human friendly.

- `--pdcp-sn-sz <n>`

  Set PDCP sequence number size(n) in bits. Valid values of n will be 5/7/12/15/18.

- `--pdcp-domain <control/user>`

  Set PDCP domain to specify Control/user plane.

Test Vector File

The test vector file is a text file contain information about test vectors. The file is made of the sections. The first section doesn't have header. It contain global information used in each test variant vectors - typically information about plaintext, ciphertext, cipher key, auth key, initial vector. All other sections begin header. The sections contain particular information typically digest.

Format of the file:

Each line beginning with sign '#' contain comment and it is ignored by parser:

```
# <comment>
```

Header line is just name in square bracket:

```
[<section name>]
```

Data line contain information token then sign '=' and a string of bytes in C byte array format:

```
<token> = <C byte array>
```

Tokens list:

- `plaintext`

  Original plaintext to be encrypted.

- `ciphertext`

  Encrypted plaintext string.

- `cipher_key`

  Key used in cipher operation.

- `auth_key`

  Key used in auth operation.

- `cipher_iv`

  Cipher Initial Vector.

- `auth_iv`

  Auth Initial Vector.

- `aad`

  Additional data.

- `digest`

  Digest string.

7.6.4 Examples

Call application for performance throughput test of single Aesni MB PMD for cipher encryption aes-cbc and auth generation sha1-hmac, one million operations, burst size 32, packet size 64:

```
dpdk-test-crypto-perf -l 6-7 --vdev crypto_aesni_mb -w 0000:00:00.0 --
--ptest throughput --devtype crypto_aesni_mb --optype cipher-then-auth
--cipher-algo aes-cbc --cipher-op encrypt --cipher-key-sz 16 --auth-algo
sha1-hmac --auth-op generate --auth-key-sz 64 --digest-sz 12
--total-ops 10000000 --burst-sz 32 --buffer-sz 64
```

Call application for performance latency test of two Aesni MB PMD executed on two cores for cipher encryption aes-cbc, ten operations in silent mode:

```
dpdk-test-crypto-perf -l 4-7 --vdev crypto_aesni_mb1
--vdev crypto_aesni_mb2 -w 0000:00:00.0 -- --devtype crypto_aesni_mb
--cipher-algo aes-cbc --cipher-key-sz 16 --cipher-iv-sz 16
--cipher-op encrypt --optype cipher-only --silent
--ptest latency --total-ops 10
```

Call application for verification test of single open ssl PMD for cipher encryption aes-gcm and auth generation aes-gcm,ten operations in silent mode, test vector provide in file "test_aes_gcm.data" with packet verification:

```
dpdk-test-crypto-perf -l 4-7 --vdev crypto_openssl -w 0000:00:00.0 --
--devtype crypto_openssl --aead-algo aes-gcm --aead-key-sz 16
--aead-iv-sz 16 --aead-op encrypt --aead-aad-sz 16 --digest-sz 16
--optype aead --silent --ptest verify --total-ops 10
--test-file test_aes_gcm.data
```

Test vector file for cipher algorithm aes cbc 256 with authorization sha:

```
# Global Section
plaintext =
0xff, 0xca, 0xfb, 0xf1, 0x38, 0x20, 0x2f, 0x7b, 0x24, 0x98, 0x26, 0x7d, 0x1d, 0x9f, 0xb3, 0x93,
0xd9, 0xef, 0xbd, 0xad, 0x4e, 0x40, 0xbd, 0x60, 0xe9, 0x48, 0x59, 0x90, 0x67, 0xd7, 0x2b, 0x7b,
0x8a, 0xe0, 0x4d, 0xb0, 0x70, 0x38, 0xcc, 0x48, 0x61, 0x7d, 0xee, 0xd6, 0x35, 0x49, 0xae, 0xb4,
0xaf, 0x6b, 0xdd, 0xe6, 0x21, 0xc0, 0x60, 0xce, 0x0a, 0xf4, 0x1c, 0x2e, 0x1c, 0x8d, 0xe8, 0x7b
ciphertext =
0x77, 0xF9, 0xF7, 0x7A, 0xA3, 0xCB, 0x68, 0x1A, 0x11, 0x70, 0xD8, 0x7A, 0xB6, 0xE2, 0x37, 0x7E,
0xD1, 0x57, 0x1C, 0x8E, 0x85, 0xD8, 0x08, 0xBF, 0x57, 0x1F, 0x21, 0x6C, 0xAD, 0xAD, 0x47, 0x1E,
0x0D, 0x6B, 0x79, 0x39, 0x15, 0x4E, 0x5B, 0x59, 0x2D, 0x76, 0x87, 0xA6, 0xD6, 0x47, 0x8F, 0x82,
0xB8, 0x51, 0x91, 0x32, 0x60, 0xCB, 0x97, 0xDE, 0xBE, 0xF0, 0xAD, 0xFC, 0x23, 0x2E, 0x22, 0x02
cipher_key =
0xE4, 0x23, 0x33, 0x8A, 0x35, 0x64, 0x61, 0xE2, 0x49, 0x03, 0xDD, 0xC6, 0xB8, 0xCA, 0x55, 0x7A,
0xd0, 0xe7, 0x4b, 0xfb, 0x5d, 0xe5, 0x0c, 0xe7, 0x6f, 0x21, 0xb5, 0x52, 0x2a, 0xbb, 0xc7, 0xf7
auth_key =
0xaf, 0x96, 0x42, 0xf1, 0x8c, 0x50, 0xdc, 0x67, 0x1a, 0x43, 0x47, 0x62, 0xc7, 0x04, 0xab, 0x05,
0xf5, 0x0c, 0xe7, 0xa2, 0xa6, 0x23, 0xd5, 0x3d, 0x95, 0xd8, 0xcd, 0x86, 0x79, 0xf5, 0x01, 0x47,
0x4f, 0xf9, 0x1d, 0x9d, 0x36, 0xf7, 0x68, 0x1a, 0x64, 0x44, 0x58, 0x5d, 0xe5, 0x81, 0x15, 0x2a,
0x41, 0xe4, 0x0e, 0xaa, 0x1f, 0x04, 0x21, 0xff, 0x2c, 0xf3, 0x73, 0x2b, 0x48, 0x1e, 0xd2, 0xf7
cipher_iv =
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F
# Section sha 1 hmac buff 32
[sha1_hmac_buff_32]
digest =
0x36, 0xCA, 0x49, 0x6A, 0xE3, 0x54, 0xD8, 0x4F, 0x0B, 0x76, 0xD8, 0xAA, 0x78, 0xEB, 0x9D, 0x65,
0x2C, 0xCA, 0x1F, 0x97
# Section sha 256 hmac buff 32
[sha256_hmac_buff_32]
```

(continues on next page)

(continued from previous page)

```
digest =
0x1C, 0xB2, 0x3D, 0xD1, 0xF9, 0xC7, 0x6C, 0x49, 0x2E, 0xDA, 0x94, 0x8B, 0xF1, 0xCF, 0x96, 0x43,
0x67, 0x50, 0x39, 0x76, 0xB5, 0xA1, 0xCE, 0xA1, 0xD7, 0x77, 0x10, 0x07, 0x43, 0x37, 0x05, 0xB4
```

7.7 dpdk-test-compress-perf Tool

The `dpdk-test-compress-perf` tool is a Data Plane Development Kit (DPDK) utility that allows measuring performance parameters of PMDs available in the compress tree. User can use multiple cores to run tests on but only one type of compression PMD can be measured during single application execution. The tool reads the data from a file (–input-file), dumps all the file into a buffer and fills out the data of input mbufs, which are passed to compress device with compression operations. Then, the output buffers are fed into the decompression stage, and the resulting data is compared against the original data (verification phase). After that, a number of iterations are performed, compressing first and decompressing later, to check the throughput rate (showing cycles/iteration, cycles/Byte and Gbps, for compression and decompression). Another option: `pmd-cyclecount`, gives the user the opportunity to measure the number of cycles per operation for the 3 phases: setup, enqueue_burst and dequeue_burst, for both compression and decompression. An optional delay can be inserted between enqueue and dequeue so no cycles are wasted in retries while waiting for a hardware device to finish. Although artificial, this allows to measure the minimum offload cost which could be achieved in a perfectly tuned system. Comparing the results of the two tests gives information about the trade-off between throughput and cycle-count.

Note: if the max-num-sgl-segs x seg_sz > input size then segments number in the chain will be lower than value passed into max-num-sgl-segs.

7.7.1 Limitations

- Stateful operation is not supported in this version.

7.7.2 EAL Options

The following are the EAL command-line options that can be used in conjunction with the `dpdk-test-compress-perf` application. See the DPDK Getting Started Guides for more information on these options.

- `-c <COREMASK>` or `-l <CORELIST>`

  Set the hexadecimal bitmask of the cores to run on. The corelist is a list cores to use.

Note: One lcore is needed for process admin, tests are run on all other cores. To run tests on two lcores, three lcores must be passed to the tool.

- `-w <PCI>`

  Add a PCI device in white list.

- `--vdev <driver><id>`

  Add a virtual device.

7.7.3 Application Options

`--ptest [throughput/verify/pmd-cyclecount]`: set test type (default: throughput)

`--driver-name NAME`: compress driver to use

`--input-file NAME`: file to compress and decompress

`--extended-input-sz N`: extend file data up to this size (default: no extension)

`--seg-sz N`: size of segment to store the data (default: 2048)

`--burst-sz N`: compress operation burst size

`--pool-sz N`: mempool size for compress operations/mbufs (default: 8192)

`--max-num-sgl-segs N`: maximum number of segments for each mbuf (default: 16)

`--num-iter N`: number of times the file will be compressed/decompressed (default: 10000)

`--operation [comp/decomp/comp_and_decomp]`: perform test on compression, decompression or both operations

`--huffman-enc [fixed/dynamic/default]`: Huffman encoding (default: dynamic)

`--compress-level N`: compression level, which could be a single value, list or range (default: range between 1 and 9)

`--window-sz N`: base two log value of compression window size (default: max supported by PMD)

`--external-mbufs`: allocate and use memzones as external buffers instead of keeping the data directly in mbuf areas

`--cc-delay-us N`: delay between enqueue and dequeue operations in microseconds, valid only for the cyclecount test (default: 500 us)

`-h`: prints this help

Compiling the Tool

Step 1: PMD setting

The `dpdk-test-compress-perf` tool depends on compression device drivers PMD which can be disabled by default in the build configuration file `common_base`. The compression device drivers PMD which should be tested can be enabled by setting e.g.:

```
CONFIG_RTE_LIBRTE_PMD_ISAL=y
```

Running the Tool

The tool has a number of command line options. Here is the sample command line:

```
./build/app/dpdk-test-compress-perf  -l 4 -- --driver-name compress_qat --input-file test.txt -
↪-seg-sz 8192
 --compress-level 1:1:9 --num-iter 10 --extended-input-sz 1048576  --max-num-sgl-segs 16 --
↪huffman-enc fixed
```

7.8 dpdk-test-eventdev Application

The `dpdk-test-eventdev` tool is a Data Plane Development Kit (DPDK) application that allows exercising various eventdev use cases. This application has a generic framework to add new eventdev based test cases to verify functionality and measure the performance parameters of DPDK eventdev devices.

7.8.1 Compiling the Application

Build the application

Execute the `dpdk-setup.sh` script to build the DPDK library together with the `dpdk-test-eventdev` application.

Initially, the user must select a DPDK target to choose the correct target type and compiler options to use when building the libraries. The user must have all libraries, modules, updates and compilers installed in the system prior to this, as described in the earlier chapters in this Getting Started Guide.

7.8.2 Running the Application

The application has a number of command line options:

```
dpdk-test-eventdev [EAL Options] -- [application options]
```

EAL Options

The following are the EAL command-line options that can be used in conjunction with the `dpdk-test-eventdev` application. See the DPDK Getting Started Guides for more information on these options.

- `-c <COREMASK>` or `-l <CORELIST>`

  Set the hexadecimal bitmask of the cores to run on. The corelist is a list of cores to use.

- `--vdev <driver><id>`

  Add a virtual eventdev device.

Application Options

The following are the application command-line options:

- `--verbose`

  Set verbose level. Default is 1. Value > 1 displays more details.

- `--dev <n>`

  Set the device id of the event device.

- `--test <name>`

  Set test name, where `name` is one of the following:

  ```
  order_queue
  order_atq
  perf_queue
  perf_atq
  pipeline_atq
  pipeline_queue
  ```

- `--socket_id <n>`

  Set the socket id of the application resources.

- `--pool-sz <n>`

  Set the number of mbufs to be allocated from the mempool.

- `--plcores <CORELIST>`

  Set the list of cores to be used as producers.

- `--wlcores <CORELIST>`

  Set the list of cores to be used as workers.

- `--stlist <type_list>`

  Set the scheduled type of each stage where `type_list` size determines the number of stages used in the test application. Each type_list member can be one of the following:

  ```
  P or p : Parallel schedule type
  O or o : Ordered schedule type
  A or a : Atomic schedule type
  ```

  Application expects the `type_list` in comma separated form (i.e. `--stlist o,a,a,a`)

- `--nb_flows <n>`

  Set the number of flows to produce.

- `--nb_pkts <n>`

  Set the number of packets to produce. 0 implies no limit.

- `--worker_deq_depth <n>`

  Set the dequeue depth of the worker.

- `--fwd_latency`

Perform forward latency measurement.

- `--queue_priority`

  Enable queue priority.

- `--prod_type_ethdev`

  Use ethernet device as producer.

- `--prod_type_timerdev`

  Use event timer adapter as producer.

- `--prod_type_timerdev_burst`

  Use burst mode event timer adapter as producer.

- `--timer_tick_nsec`

  Used to dictate number of nano seconds between bucket traversal of the event timer adapter. Refer *rte_event_timer_adapter_conf*.

- `--max_tmo_nsec`

  Used to configure event timer adapter max arm timeout in nano seconds.

- `--expiry_nsec`

  Dictate the number of nano seconds after which the event timer expires.

- `--nb_timers`

  Number of event timers each producer core will generate.

- `--nb_timer_adptrs`

  Number of event timer adapters to be used. Each adapter is used in round robin manner by the producer cores.

- `--deq_tmo_nsec`

  Global dequeue timeout for all the event ports if the provided dequeue timeout is out of the supported range of event device it will be adjusted to the highest/lowest supported dequeue timeout supported.

- `--mbuf_sz`

  Set packet mbuf size. Can be used to configure Jumbo Frames. Only applicable for *pipeline_atq* and *pipeline_queue* tests.

- `--max_pkt_sz`

  Set max packet mbuf size. Can be used configure Rx/Tx scatter gather. Only applicable for *pipeline_atq* and *pipeline_queue* tests.

7.8.3 Eventdev Tests

ORDER_QUEUE Test

This is a functional test case that aims at testing the following:

1. Verify the ingress order maintenance.
2. Verify the exclusive(atomic) access to given atomic flow per eventdev port.

Table 7.1: Order queue test eventdev configuration.

#	Items	Value	Comments
1	nb_queues	2	q0(ordered), q1(atomic)
2	nb_producers	1	
3	nb_workers	>= 1	
4	nb_ports	nb_workers + 1	Workers use port 0 to port n-1. Producer uses port n

Fig. 7.1: order queue test operation.

The order queue test configures the eventdev with two queues and an event producer to inject the events to q0(ordered) queue. Both q0(ordered) and q1(atomic) are linked to all the workers.

The event producer maintains a sequence number per flow and injects the events to the ordered queue. The worker receives the events from ordered queue and forwards to atomic queue. Since the events from an ordered queue can be processed in parallel on the different workers, the ingress order of events might have changed on the downstream atomic queue enqueue. On enqueue to the atomic queue, the eventdev PMD driver reorders the event to the original ingress order(i.e producer ingress order).

When the event is dequeued from the atomic queue by the worker, this test verifies the expected sequence number of associated event per flow by comparing the free running expected sequence number per flow.

Application options

Supported application command line options are following:

```
--verbose
--dev
--test
--socket_id
--pool_sz
--plcores
--wlcores
--nb_flows
--nb_pkts
--worker_deq_depth
--deq_tmo_nsec
```

Example

Example command to run order queue test:

```
sudo build/app/dpdk-test-eventdev --vdev=event_sw0 -- \
             --test=order_queue --plcores 1 --wlcores 2,3
```

ORDER_ATQ Test

This test verifies the same aspects of `order_queue` test, the difference is the number of queues used, this test operates on a single `all types queue(atq)` instead of two different queues for ordered and atomic.

Table 7.2: Order all types queue test eventdev configuration.

#	Items	Value	Comments
1	nb_queues	1	q0(all types queue)
2	nb_producers	1	
3	nb_workers	>= 1	
4	nb_ports	nb_workers + 1	Workers use port 0 to port n-1.Producer uses port n.

Fig. 7.2: order all types queue test operation.

Application options

Supported application command line options are following:

```
--verbose
--dev
--test
--socket_id
--pool_sz
--plcores
--wlcores
--nb_flows
--nb_pkts
--worker_deq_depth
--deq_tmo_nsec
```

Example

Example command to run order `all types queue` test:

```
sudo build/app/dpdk-test-eventdev --vdev=event_octeontx -- \
                     --test=order_atq --plcores 1 --wlcores 2,3
```

PERF_QUEUE Test

This is a performance test case that aims at testing the following:

1. Measure the number of events can be processed in a second.
2. Measure the latency to forward an event.

Table 7.3: Perf queue test eventdev configuration.

#	Items	Value	Comments
1	nb_queues	nb_producers * nb_stages	Queues will be configured based on the user requested sched type list(–stlist)
2	nb_producers	>= 1	Selected through –plcores command line argument.
3	nb_workers	>= 1	Selected through –wlcores command line argument
4	nb_ports	nb_workers + nb_producers	Workers use port 0 to port n-1. Producers use port n to port p

Fig. 7.3: perf queue test operation.

The perf queue test configures the eventdev with Q queues and P ports, where Q and P is a function of the number of workers, the number of producers and number of stages as mentioned in Table 7.3.

The user can choose the number of workers, the number of producers and number of stages through the `--wlcores`, `--plcores` and the `--stlist` application command line arguments respectively.

The producer(s) injects the events to eventdev based the first stage sched type list requested by the user through `--stlist` the command line argument.

Based on the number of stages to process(selected through `--stlist`), The application forwards the event to next upstream queue and terminates when it reaches the last stage in the pipeline. On event termination, application increments the number events processed and print periodically in one second to get the number of events processed in one second.

When `--fwd_latency` command line option selected, the application inserts the timestamp in the event on the first stage and then on termination, it updates the number of cycles to forward a packet. The application uses this value to compute the average latency to a forward packet.

When `--prod_type_ethdev` command line option is selected, the application uses the probed ethernet devices as producers by configuring them as Rx adapters instead of using synthetic producers.

Application options

Supported application command line options are following:

```
--verbose
--dev
--test
--socket_id
--pool_sz
--plcores
--wlcores
--stlist
--nb_flows
```

(continues on next page)

(continued from previous page)

```
--nb_pkts
--worker_deq_depth
--fwd_latency
--queue_priority
--prod_type_ethdev
--prod_type_timerdev_burst
--prod_type_timerdev
--timer_tick_nsec
--max_tmo_nsec
--expiry_nsec
--nb_timers
--nb_timer_adptrs
--deq_tmo_nsec
```

Example

Example command to run perf queue test:

```
sudo build/app/dpdk-test-eventdev -c 0xf -s 0x1 --vdev=event_sw0 -- \
     --test=perf_queue --plcores=2 --wlcore=3 --stlist=p --nb_pkts=0
```

Example command to run perf queue test with ethernet ports:

```
sudo build/app/dpdk-test-eventdev --vdev=event_sw0 -- \
     --test=perf_queue --plcores=2 --wlcore=3 --stlist=p --prod_type_ethdev
```

Example command to run perf queue test with event timer adapter:

```
sudo  build/app/dpdk-test-eventdev --vdev="event_octeontx" -- \
             --wlcores 4 --plcores 12 --test perf_queue --stlist=a \
             --prod_type_timerdev --fwd_latency
```

PERF_ATQ Test

This is a performance test case that aims at testing the following with `all types queue` eventdev scheme.

1. Measure the number of events can be processed in a second.
2. Measure the latency to forward an event.

Table 7.4: Perf all types queue test eventdev configuration.

#	Items	Value	Comments
1	nb_queues	nb_producers	Queues will be configured based on the user requested sched type list(–stlist)
2	nb_producers	>= 1	Selected through –plcores command line argument.
3	nb_workers	>= 1	Selected through –wlcores command line argument
4	nb_ports	nb_workers + nb_producers	Workers use port 0 to port n-1. Producers use port n to port p

Fig. 7.4: perf all types queue test operation.

The `all types queues(atq)` perf test configures the eventdev with Q queues and P ports, where Q and P is a function of the number of workers and number of producers as mentioned in Table 7.4.

The atq queue test functions as same as `perf_queue` test. The difference is, It uses, `all type queue scheme` instead of separate queues for each stage and thus reduces the number of queues required to realize the use case and enables flow pinning as the event does not move to the next queue.

Application options

Supported application command line options are following:

```
--verbose
--dev
--test
--socket_id
--pool_sz
--plcores
--wlcores
--stlist
--nb_flows
--nb_pkts
--worker_deq_depth
--fwd_latency
--prod_type_ethdev
--prod_type_timerdev_burst
--prod_type_timerdev
--timer_tick_nsec
--max_tmo_nsec
--expiry_nsec
--nb_timers
--nb_timer_adptrs
--deq_tmo_nsec
```

Example

Example command to run perf `all types queue` test:

```
sudo build/app/dpdk-test-eventdev --vdev=event_octeontx -- \
             --test=perf_atq --plcores=2 --wlcore=3 --stlist=p --nb_pkts=0
```

Example command to run perf `all types queue` test with event timer adapter:

```
sudo  build/app/dpdk-test-eventdev --vdev="event_octeontx" -- \
             --wlcores 4 --plcores 12 --test perf_atq --verbose 20 \
             --stlist=a --prod_type_timerdev --fwd_latency
```

PIPELINE_QUEUE Test

This is a pipeline test case that aims at testing the following:

1. Measure the end-to-end performance of an event dev with a ethernet dev.
2. Maintain packet ordering from Rx to Tx.

Table 7.5: Pipeline queue test eventdev configuration.

#	Items	Value	Comments
1	nb_queues	(nb_producers * nb_stages) + nb_producers	Queues will be configured based on the user requested sched type list(–stlist) At the last stage of the schedule list the event is enqueued onto per port unique queue which is then Transmitted.
2	nb_producers	Selected	Producers will be configured based on the number of detected ethernet devices. Each ethdev will be configured as an Rx adapter.
3	nb_workers	Selected	Selected through –wlcores command line argument
4	nb_ports	nb_workers + (nb_produces * 2)	Workers use port 0 to port n. Producers use port n+1 to port n+m, depending on the Rx adapter capability. Consumers use port n+m+1 to port n+o depending on the Tx adapter capability.

Fig. 7.5: pipeline queue test operation.

The pipeline queue test configures the eventdev with Q queues and P ports, where Q and P is a function of the number of workers, the number of producers and number of stages as mentioned in Table 7.5.

The user can choose the number of workers and number of stages through the `--wlcores` and the `--stlist` application command line arguments respectively.

The number of producers depends on the number of ethernet devices detected and each ethernet device is configured as a event_eth_rx_adapter that acts as a producer.

The producer(s) injects the events to eventdev based the first stage sched type list requested by the user through `--stlist` the command line argument.

Based on the number of stages to process(selected through `--stlist`), The application forwards the event to next upstream queue and when it reaches the last stage in the pipeline if the event type is `atomic` it is enqueued onto ethdev Tx queue else to maintain ordering the event type is set to `atomic` and enqueued onto the last stage queue.

If the ethdev and eventdev pair have `RTE_EVENT_ETH_TX_ADAPTER_CAP_INTERNAL_PORT` capability then the worker cores enqueue the packets to the eventdev directly using `rte_event_eth_tx_adapter_enqueue` else the worker cores enqueue the packet onto the `SINGLE_LINK_QUEUE` that is managed by the Tx adapter. The Tx adapter dequeues the packet and transmits it.

On packet Tx, application increments the number events processed and print periodically in one second to get the number of events processed in one second.

Application options

Supported application command line options are following:

```
--verbose
--dev
--test
--socket_id
--pool_sz
--wlcores
--stlist
--worker_deq_depth
--prod_type_ethdev
--deq_tmo_nsec
```

Note:

- The `--prod_type_ethdev` is mandatory for running this test.

Example

Example command to run pipeline queue test:

```
sudo build/app/dpdk-test-eventdev -c 0xf -s 0x8 --vdev=event_sw0 -- \
    --test=pipeline_queue --wlcore=1 --prod_type_ethdev --stlist=a
```

PIPELINE_ATQ Test

This is a pipeline test case that aims at testing the following with `all types queue` eventdev scheme.

1. Measure the end-to-end performance of an event dev with a ethernet dev.
2. Maintain packet ordering from Rx to Tx.

Table 7.6: Pipeline atq test eventdev configuration.

#	Items	Value	Comments
1	nb_queues	nb_producers + x	Queues will be configured based on the user requested sched type list(–stlist) where x = nb_producers in generic pipeline and 0 if all the ethdev being used have Internal port capability
2	nb_producers	ethdevs	Producers will be configured based on the number of detected ethernet devices. Each ethdev will be configured as an Rx adapter.
3	nb_workers	wlcores	Selected through –wlcores command line argument
4	nb_ports	nb_workers + nb_producers + x	Workers use port 0 to port n. Producers use port n+1 to port n+m, depending on the Rx adapter capability. x = nb_producers in generic pipeline and 0 if all the ethdev being used have Internal port capability. Consumers may use port n+m+1 to port n+o depending on the Tx adapter capability.

Fig. 7.6: pipeline atq test operation.

The pipeline atq test configures the eventdev with Q queues and P ports, where Q and P is a function of the number of workers, the number of producers and number of stages as mentioned in Table 7.6.

The atq queue test functions as same as `pipeline_queue` test. The difference is, It uses, `all type queue scheme` instead of separate queues for each stage and thus reduces the number of queues required to realize the use case.

Application options

Supported application command line options are following:

```
--verbose
--dev
--test
--socket_id
--pool_sz
--wlcores
--stlist
--worker_deq_depth
--prod_type_ethdev
--deq_tmo_nsec
```

Note:

- The `--prod_type_ethdev` is mandatory for running this test.

Example

Example command to run pipeline queue test:

```
sudo build/app/dpdk-test-eventdev -c 0xf -s 0x8 --vdev=event_sw0 -- \
    --test=pipeline_atq --wlcore=1 --prod_type_ethdev --stlist=a
```

CHAPTER

EIGHT

TESTPMD APPLICATION USER GUIDE

8.1 Introduction

This document is a user guide for the `testpmd` example application that is shipped as part of the Data Plane Development Kit.

The `testpmd` application can be used to test the DPDK in a packet forwarding mode and also to access NIC hardware features such as Flow Director. It also serves as a example of how to build a more fully-featured application using the DPDK SDK.

The guide shows how to build and run the testpmd application and how to configure the application from the command line and the run-time environment.

8.2 Compiling the Application

The `testpmd` application is compiled as part of the main compilation of the DPDK libraries and tools. Refer to the DPDK Getting Started Guides for details. The basic compilation steps are:

1. Set the required environmental variables and go to the source directory:

```
export RTE_SDK=/path/to/rte_sdk
cd $RTE_SDK
```

2. Set the compilation target. For example:

```
export RTE_TARGET=x86_64-native-linux-gcc
```

3. Build the application:

```
make install T=$RTE_TARGET
```

   The compiled application will be located at:

```
$RTE_SDK/$RTE_TARGET/app/testpmd
```

8.3 Running the Application

8.3.1 EAL Command-line Options

Please refer to *EAL parameters (Linux)* or *EAL parameters (FreeBSD)* for a list of available EAL command-line options.

8.3.2 Testpmd Command-line Options

The following are the command-line options for the testpmd applications. They must be separated from the EAL options, shown in the previous section, with a `--` separator:

```
sudo ./testpmd -l 0-3 -n 4 -- -i --portmask=0x1 --nb-cores=2
```

The command line options are:

- `-i, --interactive`

  Run testpmd in interactive mode. In this mode, the testpmd starts with a prompt that can be used to start and stop forwarding, configure the application and display stats on the current packet processing session. See *Testpmd Runtime Functions* for more details.

  In non-interactive mode, the application starts with the configuration specified on the command-line and immediately enters forwarding mode.

- `-h, --help`

  Display a help message and quit.

- `-a, --auto-start`

  Start forwarding on initialization.

- `--tx-first`

  Start forwarding, after sending a burst of packets first.

Note: This flag should be only used in non-interactive mode.

- `--stats-period PERIOD`

  Display statistics every PERIOD seconds, if interactive mode is disabled. The default value is 0, which means that the statistics will not be displayed.

- `--nb-cores=N`

  Set the number of forwarding cores, where 1 <= N <= "number of cores" or `CONFIG_RTE_MAX_LCORE` from the configuration file. The default value is 1.

- `--nb-ports=N`

  Set the number of forwarding ports, where 1 <= N <= "number of ports" on the board or `CONFIG_RTE_MAX_ETHPORTS` from the configuration file. The default value is the number of ports on the board.

- `--coremask=0xXX`

  Set the hexadecimal bitmask of the cores running the packet forwarding test. The master lcore is reserved for command line parsing only and cannot be masked on for packet forwarding.

- `--portmask=0xXX`

  Set the hexadecimal bitmask of the ports used by the packet forwarding test.

- `--portlist=X`

  Set the forwarding ports based on the user input used by the packet forwarding test. '-' denotes a range of ports to set including the two specified port IDs ',' separates multiple port values. Possible examples like –portlist=0,1 or –portlist=0-2 or –portlist=0,1-2 etc

- `--numa`

  Enable NUMA-aware allocation of RX/TX rings and of RX memory buffers (mbufs). [Default setting]

- `--no-numa`

  Disable NUMA-aware allocation of RX/TX rings and of RX memory buffers (mbufs).

- `--port-numa-config=(port,socket)[,(port,socket)]`

  Specify the socket on which the memory pool to be used by the port will be allocated.

- `--ring-numa-config=(port,flag,socket)[,(port,flag,socket)]`

  Specify the socket on which the TX/RX rings for the port will be allocated. Where flag is 1 for RX, 2 for TX, and 3 for RX and TX.

- `--socket-num=N`

  Set the socket from which all memory is allocated in NUMA mode, where 0 <= N < number of sockets on the board.

- `--mbuf-size=N`

  Set the data size of the mbufs used to N bytes, where N < 65536. The default value is 2048.

- `--total-num-mbufs=N`

  Set the number of mbufs to be allocated in the mbuf pools, where N > 1024.

- `--max-pkt-len=N`

  Set the maximum packet size to N bytes, where N >= 64. The default value is 1518.

- `--max-lro-pkt-size=N`

  Set the maximum LRO aggregated packet size to N bytes, where N >= 64.

- `--eth-peers-configfile=name`

  Use a configuration file containing the Ethernet addresses of the peer ports. The configuration file should contain the Ethernet addresses on separate lines:

```
XX:XX:XX:XX:XX:01
XX:XX:XX:XX:XX:02
...
```

- `--eth-peer=N,XX:XX:XX:XX:XX:XX`

  Set the MAC address `XX:XX:XX:XX:XX:XX` of the peer port N, where 0 <= N < `CONFIG_RTE_MAX_ETHPORTS` from the configuration file.

- `--tx-ip=SRC,DST`

  Set the source and destination IP address used when doing transmit only test. The defaults address values are source 198.18.0.1 and destination 198.18.0.2. These are special purpose addresses reserved for benchmarking (RFC 5735).

- `--tx-udp=SRC[,DST]`

  Set the source and destination UDP port number for transmit test only test. The default port is the port 9 which is defined for the discard protocol (RFC 863).

- `--pkt-filter-mode=mode`

  Set Flow Director mode where mode is either `none` (the default), `signature` or `perfect`. See *flow_director_filter* for more details.

- `--pkt-filter-report-hash=mode`

  Set Flow Director hash match reporting mode where mode is `none`, `match` (the default) or `always`.

- `--pkt-filter-size=N`

  Set Flow Director allocated memory size, where N is 64K, 128K or 256K. Sizes are in kilobytes. The default is 64.

- `--pkt-filter-flexbytes-offset=N`

  Set the flexbytes offset. The offset is defined in words (not bytes) counted from the first byte of the destination Ethernet MAC address, where N is 0 <= N <= 32. The default value is 0x6.

- `--pkt-filter-drop-queue=N`

  Set the drop-queue. In perfect filter mode, when a rule is added with queue = -1, the packet will be enqueued into the RX drop-queue. If the drop-queue does not exist, the packet is dropped. The default value is N=127.

- `--disable-crc-strip`

  Disable hardware CRC stripping.

- `--enable-lro`

  Enable large receive offload.

- `--enable-rx-cksum`

  Enable hardware RX checksum offload.

- `--enable-scatter`

  Enable scatter (multi-segment) RX.

- `--enable-hw-vlan`

  Enable hardware VLAN.

- `--enable-hw-vlan-filter`

  Enable hardware VLAN filter.

- `--enable-hw-vlan-strip`

  Enable hardware VLAN strip.

- `--enable-hw-vlan-extend`

  Enable hardware VLAN extend.

- `--enable-hw-qinq-strip`

  Enable hardware QINQ strip.

- `--enable-drop-en`

  Enable per-queue packet drop for packets with no descriptors.

- `--disable-rss`

  Disable RSS (Receive Side Scaling).

- `--port-topology=mode`

  Set port topology, where mode is `paired` (the default), `chained` or `loop`.

  In `paired` mode, the forwarding is between pairs of ports, for example: (0,1), (2,3), (4,5).

  In `chained` mode, the forwarding is to the next available port in the port mask, for example: (0,1), (1,2), (2,0).

  The ordering of the ports can be changed using the portlist testpmd runtime function.

  In `loop` mode, ingress traffic is simply transmitted back on the same interface.

- `--forward-mode=mode`

  Set the forwarding mode where `mode` is one of the following:

```
io (the default)
mac
macswap
flowgen
rxonly
txonly
csum
icmpecho
ieee1588
tm
noisy
```

- `--rss-ip`

  Set RSS functions for IPv4/IPv6 only.

- `--rss-udp`

  Set RSS functions for IPv4/IPv6 and UDP.

- `--rxq=N`

  Set the number of RX queues per port to N, where 1 <= N <= 65535. The default value is 1.

- `--rxd=N`

  Set the number of descriptors in the RX rings to N, where N > 0. The default value is 128.

- `--txq=N`

  Set the number of TX queues per port to N, where 1 <= N <= 65535. The default value is 1.

- `--txd=N`

  Set the number of descriptors in the TX rings to N, where N > 0. The default value is 512.

- `--hairpinq=N`

  Set the number of hairpin queues per port to N, where 1 <= N <= 65535. The default value is 0. The number of hairpin queues are added to the number of TX queues and to the number of RX queues. then the first RX hairpin is binded to the first TX hairpin, the second RX hairpin is binded to the second TX hairpin and so on. The index of the first RX hairpin queue is the number of RX queues as configured using –rxq. The index of the first TX hairpin queue is the number of TX queues as configured using –txq.

- `--burst=N`

  Set the number of packets per burst to N, where 1 <= N <= 512. The default value is 32. If set to 0, driver default is used if defined. Else, if driver default is not defined, default of 32 is used.

- `--mbcache=N`

  Set the cache of mbuf memory pools to N, where 0 <= N <= 512. The default value is 16.

- `--rxpt=N`

  Set the prefetch threshold register of RX rings to N, where N >= 0. The default value is 8.

- `--rxht=N`

  Set the host threshold register of RX rings to N, where N >= 0. The default value is 8.

- `--rxfreet=N`

  Set the free threshold of RX descriptors to N, where 0 <= N < value of –rxd. The default value is 0.

- `--rxwt=N`

  Set the write-back threshold register of RX rings to N, where N >= 0. The default value is 4.

- `--txpt=N`

  Set the prefetch threshold register of TX rings to N, where N >= 0. The default value is 36.

- `--txht=N`

  Set the host threshold register of TX rings to N, where N >= 0. The default value is 0.

- `--txwt=N`

  Set the write-back threshold register of TX rings to N, where N >= 0. The default value is 0.

- `--txfreet=N`

  Set the transmit free threshold of TX rings to N, where 0 <= N <= value of `--txd`. The default value is 0.

- `--txrst=N`

  Set the transmit RS bit threshold of TX rings to N, where 0 <= N <= value of `--txd`. The default value is 0.

- `--rx-queue-stats-mapping=(port,queue,mapping)[,(port,queue,mapping)]`

  Set the RX queues statistics counters mapping 0 <= mapping <= 15.

- `--tx-queue-stats-mapping=(port,queue,mapping)[,(port,queue,mapping)]`

  Set the TX queues statistics counters mapping 0 <= mapping <= 15.

- `--no-flush-rx`

  Don't flush the RX streams before starting forwarding. Used mainly with the PCAP PMD.

- `--txpkts=X[,Y]`

  Set TX segment sizes or total packet length. Valid for `tx-only` and `flowgen` forwarding modes.

- `--txonly-multi-flow`

  Generate multiple flows in txonly mode.

- `--disable-link-check`

  Disable check on link status when starting/stopping ports.

- `--disable-device-start`

  Do not automatically start all ports. This allows testing configuration of rx and tx queues before device is started for the first time.

- `--no-lsc-interrupt`

  Disable LSC interrupts for all ports, even those supporting it.

- `--no-rmv-interrupt`

  Disable RMV interrupts for all ports, even those supporting it.

- `--bitrate-stats=N`

  Set the logical core N to perform bitrate calculation.

- `--print-event <unknown|intr_lsc|queue_state|intr_reset|vf_mbox|macsec|intr_rmv|dev_prob`

  Enable printing the occurrence of the designated event. Using all will enable all of them.

- `--mask-event <unknown|intr_lsc|queue_state|intr_reset|vf_mbox|macsec|intr_rmv|dev_probe`

  Disable printing the occurrence of the designated event. Using all will disable all of them.

- `--flow-isolate-all`

  Providing this parameter requests flow API isolated mode on all ports at initialization time. It ensures all traffic is received through the configured flow rules only (see flow command).

  Ports that do not support this mode are automatically discarded.

- `--tx-offloads=0xXXXXXXXX`

  Set the hexadecimal bitmask of TX queue offloads. The default value is 0.

- `--rx-offloads=0xXXXXXXXX`

  Set the hexadecimal bitmask of RX queue offloads. The default value is 0.

- `--hot-plug`

  Enable device event monitor mechanism for hotplug.

- `--vxlan-gpe-port=N`

  Set the UDP port number of tunnel VXLAN-GPE to N. The default value is 4790.

- `--mlockall`

  Enable locking all memory.

- `--no-mlockall`

  Disable locking all memory.

- `--mp-alloc <native|anon|xmem|xmemhuge>`

  Select mempool allocation mode:

  - native: create and populate mempool using native DPDK memory
  - anon: create mempool using native DPDK memory, but populate using anonymous memory
  - xmem: create and populate mempool using externally and anonymously allocated area
  - xmemhuge: create and populate mempool using externally and anonymously allocated hugepage area

- `--noisy-tx-sw-buffer-size`

  Set the number of maximum elements of the FIFO queue to be created for buffering packets. Only available with the noisy forwarding mode. The default value is 0.

- `--noisy-tx-sw-buffer-flushtime=N`

  Set the time before packets in the FIFO queue is flushed. Only available with the noisy forwarding mode. The default value is 0.

- `--noisy-lkup-memory=N`

  Set the size of the noisy neighbor simulation memory buffer in MB to N. Only available with the noisy forwarding mode. The default value is 0.

- `--noisy-lkup-num-reads=N`

  Set the number of reads to be done in noisy neighbor simulation memory buffer to N. Only available with the noisy forwarding mode. The default value is 0.

- `--noisy-lkup-num-writes=N`

  Set the number of writes to be done in noisy neighbor simulation memory buffer to N. Only available with the noisy forwarding mode. The default value is 0.

- `--noisy-lkup-num-reads-writes=N`

  Set the number of r/w accesses to be done in noisy neighbor simulation memory buffer to N. Only available with the noisy forwarding mode. The default value is 0.

- `--no-iova-contig`

  Enable to create mempool which is not IOVA contiguous. Valid only with –mp-alloc=anon. The default value is 0.

- `--rx-mq-mode`

  Set the hexadecimal bitmask of RX multi queue mode which can be enabled. The default value is 0x7:

```
ETH_MQ_RX_RSS_FLAG | ETH_MQ_RX_DCB_FLAG | ETH_MQ_RX_VMDQ_FLAG
```

8.4 Testpmd Runtime Functions

Where the testpmd application is started in interactive mode, (`-i|--interactive`), it displays a prompt that can be used to start and stop forwarding, configure the application, display statistics (including the extended NIC statistics aka xstats) , set the Flow Director and other tasks:

```
testpmd>
```

The testpmd prompt has some, limited, readline support. Common bash command-line functions such as `Ctrl+a` and `Ctrl+e` to go to the start and end of the prompt line are supported as well as access to the command history via the up-arrow.

There is also support for tab completion. If you type a partial command and hit <TAB> you get a list of the available completions:

```
testpmd> show port <TAB>

    info [Mul-choice STRING]: show|clear port info|stats|xstats|fdir|stat_qmap|dcb_tc|cap X
    info [Mul-choice STRING]: show|clear port info|stats|xstats|fdir|stat_qmap|dcb_tc|cap all
    stats [Mul-choice STRING]: show|clear port info|stats|xstats|fdir|stat_qmap|dcb_tc|cap X
    stats [Mul-choice STRING]: show|clear port info|stats|xstats|fdir|stat_qmap|dcb_tc|cap all
    ...
```

Note: Some examples in this document are too long to fit on one line are shown wrapped at "\" for display purposes:

```
testpmd> set flow_ctrl rx (on|off) tx (on|off) (high_water) (low_water) \
         (pause_time) (send_xon) (port_id)
```

In the real `testpmd>` prompt these commands should be on a single line.

8.4.1 Help Functions

The testpmd has on-line help for the functions that are available at runtime. These are divided into sections and can be accessed using help, help section or help all:

```
testpmd> help

    help control    : Start and stop forwarding.
    help display    : Displaying port, stats and config information.
    help config     : Configuration information.
    help ports      : Configuring ports.
    help registers  : Reading and setting port registers.
    help filters    : Filters configuration help.
    help all        : All of the above sections.
```

8.4.2 Command File Functions

To facilitate loading large number of commands or to avoid cutting and pasting where not practical or possible testpmd supports alternative methods for executing commands.

- If started with the `--cmdline-file=FILENAME` command line argument testpmd will execute all CLI commands contained within the file immediately before starting packet forwarding or entering interactive mode.

```
./testpmd -n4 -r2 ... -- -i --cmdline-file=/home/ubuntu/flow-create-commands.txt
Interactive-mode selected
CLI commands to be read from /home/ubuntu/flow-create-commands.txt
Configuring Port 0 (socket 0)
Port 0: 7C:FE:90:CB:74:CE
Configuring Port 1 (socket 0)
Port 1: 7C:FE:90:CB:74:CA
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
Flow rule #0 created
Flow rule #1 created
...
...
Flow rule #498 created
Flow rule #499 created
Read all CLI commands from /home/ubuntu/flow-create-commands.txt
testpmd>
```

- At run-time additional commands can be loaded in bulk by invoking the `load FILENAME` command.

```
testpmd> load /home/ubuntu/flow-create-commands.txt
Flow rule #0 created
Flow rule #1 created
...
...
Flow rule #498 created
Flow rule #499 created
Read all CLI commands from /home/ubuntu/flow-create-commands.txt
testpmd>
```

In all cases output from any included command will be displayed as standard output. Execution will continue until the end of the file is reached regardless of whether any errors occur. The end user must examine the output to determine if any failures occurred.

8.4.3 Control Functions

start

Start packet forwarding with current configuration:

```
testpmd> start
```

start tx_first

Start packet forwarding with current configuration after sending specified number of bursts of packets:

```
testpmd> start tx_first (""|burst_num)
```

The default burst number is 1 when `burst_num` not presented.

stop

Stop packet forwarding, and display accumulated statistics:

```
testpmd> stop
```

quit

Quit to prompt:

```
testpmd> quit
```

8.4.4 Display Functions

The functions in the following sections are used to display information about the testpmd configuration or the NIC status.

show port

Display information for a given port or all ports:

```
testpmd> show port (info|summary|stats|xstats|fdir|stat_qmap|dcb_tc|cap) (port_id|all)
```

The available information categories are:

- `info`: General port information such as MAC address.
- `summary`: Brief port summary such as Device Name, Driver Name etc.
- `stats`: RX/TX statistics.
- `xstats`: RX/TX extended NIC statistics.
- `fdir`: Flow Director information and statistics.
- `stat_qmap`: Queue statistics mapping.
- `dcb_tc`: DCB information such as TC mapping.
- `cap`: Supported offload capabilities.

For example:

```
testpmd> show port info 0

********************* Infos for port 0 *********************

MAC address: XX:XX:XX:XX:XX:XX
Connect to socket: 0
memory allocation on the socket: 0
Link status: up
Link speed: 40000 Mbps
Link duplex: full-duplex
Promiscuous mode: enabled
Allmulticast mode: disabled
Maximum number of MAC addresses: 64
Maximum number of MAC addresses of hash filtering: 0
VLAN offload:
    strip on, filter on, extend off, qinq strip off
Redirection table size: 512
Supported flow types:
  ipv4-frag
  ipv4-tcp
  ipv4-udp
  ipv4-sctp
  ipv4-other
  ipv6-frag
  ipv6-tcp
  ipv6-udp
  ipv6-sctp
  ipv6-other
  l2_payload
  port
  vxlan
  geneve
  nvgre
  vxlan-gpe
```

show port rss reta

Display the rss redirection table entry indicated by masks on port X:

```
testpmd> show port (port_id) rss reta (size) (mask0, mask1...)
```

size is used to indicate the hardware supported reta size

show port rss-hash

Display the RSS hash functions and RSS hash key of a port:

```
testpmd> show port (port_id) rss-hash [key]
```

clear port

Clear the port statistics and forward engine statistics for a given port or for all ports:

```
testpmd> clear port (info|stats|xstats|fdir|stat_qmap) (port_id|all)
```

For example:

```
testpmd> clear port stats all
```

show (rxq|txq)

Display information for a given port's RX/TX queue:

```
testpmd> show (rxq|txq) info (port_id) (queue_id)
```

show desc status(rxq|txq)

Display information for a given port's RX/TX descriptor status:

```
testpmd> show port (port_id) (rxq|txq) (queue_id) desc (desc_id) status
```

show config

Displays the configuration of the application. The configuration comes from the command-line, the runtime or the application defaults:

```
testpmd> show config (rxtx|cores|fwd|txpkts)
```

The available information categories are:

- `rxtx`: RX/TX configuration items.
- `cores`: List of forwarding cores.
- `fwd`: Packet forwarding configuration.
- `txpkts`: Packets to TX configuration.

For example:

```
testpmd> show config rxtx

io packet forwarding - CRC stripping disabled - packets/burst=16
nb forwarding cores=2 - nb forwarding ports=1
RX queues=1 - RX desc=128 - RX free threshold=0
```

(continues on next page)

(continued from previous page)

```
RX threshold registers: pthresh=8 hthresh=8 wthresh=4
TX queues=1 - TX desc=512 - TX free threshold=0
TX threshold registers: pthresh=36 hthresh=0 wthresh=0
TX RS bit threshold=0 - TXQ flags=0x0
```

set fwd

Set the packet forwarding mode:

```
testpmd> set fwd (io|mac|macswap|flowgen| \
                  rxonly|txonly|csum|icmpecho|noisy) (""|retry)
```

`retry` can be specified for forwarding engines except `rx_only`.

The available information categories are:

- `io`: Forwards packets "as-is" in I/O mode. This is the fastest possible forwarding operation as it does not access packets data. This is the default mode.
- `mac`: Changes the source and the destination Ethernet addresses of packets before forwarding them. Default application behavior is to set source Ethernet address to that of the transmitting interface, and destination address to a dummy value (set during init). The user may specify a target destination Ethernet address via the 'eth-peer' or 'eth-peers-configfile' command-line options. It is not currently possible to specify a specific source Ethernet address.
- `macswap`: MAC swap forwarding mode. Swaps the source and the destination Ethernet addresses of packets before forwarding them.
- `flowgen`: Multi-flow generation mode. Originates a number of flows (with varying destination IP addresses), and terminate receive traffic.
- `rxonly`: Receives packets but doesn't transmit them.
- `txonly`: Generates and transmits packets without receiving any.
- `csum`: Changes the checksum field with hardware or software methods depending on the offload flags on the packet.
- `icmpecho`: Receives a burst of packets, lookup for ICMP echo requests and, if any, send back ICMP echo replies.
- `ieee1588`: Demonstrate L2 IEEE1588 V2 PTP timestamping for RX and TX. Requires `CONFIG_RTE_LIBRTE_IEEE1588=y`.
- `softnic`: Demonstrates the softnic forwarding operation. In this mode, packet forwarding is similar to I/O mode except for the fact that packets are loopback to the softnic ports only. Therefore, portmask parameter should be set to softnic port only. The various software based custom NIC pipelines specified through the softnic firmware (DPDK packet framework script) can be tested in this mode. Furthermore, it allows to build 5-level hierarchical QoS scheduler as a default option that can be enabled through CLI once testpmd application is initialised. The user can modify the default scheduler hierarchy or can specify the new QoS Scheduler hierarchy through CLI. Requires `CONFIG_RTE_LIBRTE_PMD_SOFTNIC=y`.
- `noisy`: Noisy neighbor simulation. Simulate more realistic behavior of a guest machine engaged in receiving and sending packets performing Virtual Network Function (VNF).

Example:

```
testpmd> set fwd rxonly

Set rxonly packet forwarding mode
```

show fwd

When running, forwarding engines maintain statistics from the time they have been started. Example for the io forwarding engine, with some packet drops on the tx side:

```
testpmd> show fwd stats all

  ------- Forward Stats for RX Port= 0/Queue= 0 -> TX Port= 1/Queue= 0 -------
  RX-packets: 274293770      TX-packets: 274293642      TX-dropped: 128

  ------- Forward Stats for RX Port= 1/Queue= 0 -> TX Port= 0/Queue= 0 -------
  RX-packets: 274301850      TX-packets: 274301850      TX-dropped: 0

  ---------------------- Forward statistics for port 0  ----------------------
  RX-packets: 274293802      RX-dropped: 0             RX-total: 274293802
  TX-packets: 274301862      TX-dropped: 0             TX-total: 274301862
  ----------------------------------------------------------------------------

  ---------------------- Forward statistics for port 1  ----------------------
  RX-packets: 274301894      RX-dropped: 0             RX-total: 274301894
  TX-packets: 274293706      TX-dropped: 128           TX-total: 274293834
  ----------------------------------------------------------------------------

  ++++++++++++++++ Accumulated forward statistics for all ports++++++++++++++++
  RX-packets: 548595696      RX-dropped: 0             RX-total: 548595696
  TX-packets: 548595568      TX-dropped: 128           TX-total: 548595696
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Note: Enabling CONFIG_RTE_TEST_PMD_RECORD_CORE_CYCLES appends "CPU cycles/packet" stats, like:

CPU cycles/packet=xx.dd (total cycles=xxxx / total RX packets=xxxx) at xxx MHz clock

clear fwd

Clear the forwarding engines statistics:

```
testpmd> clear fwd stats all
```

read rxd

Display an RX descriptor for a port RX queue:

```
testpmd> read rxd (port_id) (queue_id) (rxd_id)
```

For example:

```
testpmd> read rxd 0 0 4
     0x0000000B - 0x001D0180 / 0x0000000B - 0x001D0180
```

read txd

Display a TX descriptor for a port TX queue:

```
testpmd> read txd (port_id) (queue_id) (txd_id)
```

For example:

```
testpmd> read txd 0 0 4
     0x00000001 - 0x24C3C440 / 0x000F0000 - 0x2330003C
```

ddp get list

Get loaded dynamic device personalization (DDP) package info list:

```
testpmd> ddp get list (port_id)
```

ddp get info

Display information about dynamic device personalization (DDP) profile:

```
testpmd> ddp get info (profile_path)
```

show vf stats

Display VF statistics:

```
testpmd> show vf stats (port_id) (vf_id)
```

clear vf stats

Reset VF statistics:

```
testpmd> clear vf stats (port_id) (vf_id)
```

show port pctype mapping

List all items from the pctype mapping table:

```
testpmd> show port (port_id) pctype mapping
```

show rx offloading capabilities

List all per queue and per port Rx offloading capabilities of a port:

```
testpmd> show port (port_id) rx_offload capabilities
```

show rx offloading configuration

List port level and all queue level Rx offloading configuration:

```
testpmd> show port (port_id) rx_offload configuration
```

show tx offloading capabilities

List all per queue and per port Tx offloading capabilities of a port:

```
testpmd> show port (port_id) tx_offload capabilities
```

show tx offloading configuration

List port level and all queue level Tx offloading configuration:

```
testpmd> show port (port_id) tx_offload configuration
```

show tx metadata setting

Show Tx metadata value set for a specific port:

```
testpmd> show port (port_id) tx_metadata
```

show port supported ptypes

Show ptypes supported for a specific port:

```
testpmd> show port (port_id) ptypes
```

set port supported ptypes

set packet types classification for a specific port:

```
testpmd> set port (port_id) ptypes_mask (mask)
```

show port mac addresses info

Show mac addresses added for a specific port:

```
testpmd> show port (port_id) macs
```

show port multicast mac addresses info

Show multicast mac addresses added for a specific port:

```
testpmd> show port (port_id) mcast_macs
```

show device info

Show general information about devices probed:

```
testpmd> show device info (<identifier>|all)
```

For example:

```
testpmd> show device info net_pcap0

********************* Infos for device net_pcap0 *********************
Bus name: vdev
Driver name: net_pcap
Devargs: iface=enP2p6s0,phy_mac=1
Connect to socket: -1

        Port id: 2
        MAC address: 1E:37:93:28:04:B8
        Device name: net_pcap0
```

dump physmem

Dumps all physical memory segment layouts:

```
testpmd> dump_physmem
```

dump memzone

Dumps the layout of all memory zones:

```
testpmd> dump_memzone
```

dump socket memory

Dumps the memory usage of all sockets:

```
testpmd> dump_socket_mem
```

dump struct size

Dumps the size of all memory structures:

```
testpmd> dump_struct_sizes
```

dump ring

Dumps the status of all or specific element in DPDK rings:

```
testpmd> dump_ring [ring_name]
```

dump mempool

Dumps the statistics of all or specific memory pool:

```
testpmd> dump_mempool [mempool_name]
```

dump devargs

Dumps the user device list:

```
testpmd> dump_devargs
```

dump log types

Dumps the log level for all the dpdk modules:

```
testpmd> dump_log_types
```

show (raw_encap|raw_decap)

Display content of raw_encap/raw_decap buffers in hex:

```
testpmd> show <raw_encap|raw_decap> <index>
testpmd> show <raw_encap|raw_decap> all
```

For example:

```
testpmd> show raw_encap 6

index: 6 at [0x1c565b0], len=50
00000000: 00 00 00 00 00 00 16 26 36 46 56 66 08 00 45 00 | .......&6FVf..E.
00000010: 00 00 00 00 00 00 00 11 00 00 C0 A8 01 06 C0 A8 | ................
00000020: 03 06 00 00 00 FA 00 00 00 00 08 00 00 00 00 00 | ................
00000030: 06 00                                           | ..
```

8.4.5 Configuration Functions

The testpmd application can be configured from the runtime as well as from the command-line.

This section details the available configuration functions that are available.

Note: Configuration changes only become active when forwarding is started/restarted.

---

set default

Reset forwarding to the default configuration:

```
testpmd> set default
```

set verbose

Set the debug verbosity level:

```
testpmd> set verbose (level)
```

Available levels are as following:

- `0` silent except for error.
- `1` fully verbose except for Tx packets.
- `2` fully verbose except for Rx packets.
- `> 2` fully verbose.

set log

Set the log level for a log type:

```
testpmd> set log global|(type) (level)
```

Where:

- `type` is the log name.
- `level` is the log level.

For example, to change the global log level:

```
testpmd> set log global (level)
```

Regexes can also be used for type. To change log level of user1, user2 and user3:

```
testpmd> set log user[1-3] (level)
```

set nbport

Set the number of ports used by the application:

set nbport (num)

This is equivalent to the `--nb-ports` command-line option.

set nbcore

Set the number of cores used by the application:

```
testpmd> set nbcore (num)
```

This is equivalent to the `--nb-cores` command-line option.

Note: The number of cores used must not be greater than number of ports used multiplied by the number of queues per port.

---

set coremask

Set the forwarding cores hexadecimal mask:

```
testpmd> set coremask (mask)
```

This is equivalent to the `--coremask` command-line option.

Note: The master lcore is reserved for command line parsing only and cannot be masked on for packet forwarding.

---

set portmask

Set the forwarding ports hexadecimal mask:

```
testpmd> set portmask (mask)
```

This is equivalent to the `--portmask` command-line option.

set burst

Set number of packets per burst:

```
testpmd> set burst (num)
```

This is equivalent to the `--burst command-line` option.

When retry is enabled, the transmit delay time and number of retries can also be set:

```
testpmd> set burst tx delay (microseconds) retry (num)
```

set txpkts

Set the length of each segment of the TX-ONLY packets or length of packet for FLOWGEN mode:

```
testpmd> set txpkts (x[,y]*)
```

Where x[,y]* represents a CSV list of values, without white space.

set txsplit

Set the split policy for the TX packets, applicable for TX-ONLY and CSUM forwarding modes:

```
testpmd> set txsplit (off|on|rand)
```

Where:

- `off` disable packet copy & split for CSUM mode.
- `on` split outgoing packet into multiple segments. Size of each segment and number of segments per packet is determined by `set txpkts` command (see above).
- `rand` same as 'on', but number of segments per each packet is a random value between 1 and total number of segments.

set corelist

Set the list of forwarding cores:

```
testpmd> set corelist (x[,y]*)
```

For example, to change the forwarding cores:

```
testpmd> set corelist 3,1
testpmd> show config fwd

io packet forwarding - ports=2 - cores=2 - streams=2 - NUMA support disabled
Logical Core 3 (socket 0) forwards packets on 1 streams:
RX P=0/Q=0 (socket 0) -> TX P=1/Q=0 (socket 0) peer=02:00:00:00:00:01
Logical Core 1 (socket 0) forwards packets on 1 streams:
RX P=1/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer=02:00:00:00:00:00
```

Note: The cores are used in the same order as specified on the command line.

set portlist

Set the list of forwarding ports:

```
testpmd> set portlist (x[,y]*)
```

For example, to change the port forwarding:

```
testpmd> set portlist 0,2,1,3
testpmd> show config fwd

io packet forwarding - ports=4 - cores=1 - streams=4
Logical Core 3 (socket 0) forwards packets on 4 streams:
RX P=0/Q=0 (socket 0) -> TX P=2/Q=0 (socket 0) peer=02:00:00:00:00:01
RX P=2/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer=02:00:00:00:00:00
RX P=1/Q=0 (socket 0) -> TX P=3/Q=0 (socket 0) peer=02:00:00:00:00:03
RX P=3/Q=0 (socket 0) -> TX P=1/Q=0 (socket 0) peer=02:00:00:00:00:02
```

set port setup on

Select how to retrieve new ports created after "port attach" command:

```
testpmd> set port setup on (iterator|event)
```

For each new port, a setup is done. It will find the probed ports via RTE_ETH_FOREACH_MATCHING_DEV loop in iterator mode, or via RTE_ETH_EVENT_NEW in event mode.

set tx loopback

Enable/disable tx loopback:

```
testpmd> set tx loopback (port_id) (on|off)
```

set drop enable

set drop enable bit for all queues:

```
testpmd> set all queues drop (port_id) (on|off)
```

set split drop enable (for VF)

set split drop enable bit for VF from PF:

```
testpmd> set vf split drop (port_id) (vf_id) (on|off)
```

set mac antispoof (for VF)

Set mac antispoof for a VF from the PF:

```
testpmd> set vf mac antispoof  (port_id) (vf_id) (on|off)
```

set macsec offload

Enable/disable MACsec offload:

```
testpmd> set macsec offload (port_id) on encrypt (on|off) replay-protect (on|off)
testpmd> set macsec offload (port_id) off
```

set macsec sc

Configure MACsec secure connection (SC):

```
testpmd> set macsec sc (tx|rx) (port_id) (mac) (pi)
```

Note: The pi argument is ignored for tx. Check the NIC Datasheet for hardware limits.

set macsec sa

Configure MACsec secure association (SA):

```
testpmd> set macsec sa (tx|rx) (port_id) (idx) (an) (pn) (key)
```

Note: The IDX value must be 0 or 1. Check the NIC Datasheet for hardware limits.

set broadcast mode (for VF)

Set broadcast mode for a VF from the PF:

```
testpmd> set vf broadcast (port_id) (vf_id) (on|off)
```

vlan set stripq

Set the VLAN strip for a queue on a port:

```
testpmd> vlan set stripq (on|off) (port_id,queue_id)
```

vlan set stripq (for VF)

Set VLAN strip for all queues in a pool for a VF from the PF:

```
testpmd> set vf vlan stripq (port_id) (vf_id) (on|off)
```

vlan set insert (for VF)

Set VLAN insert for a VF from the PF:

```
testpmd> set vf vlan insert (port_id) (vf_id) (vlan_id)
```

vlan set tag (for VF)

Set VLAN tag for a VF from the PF:

```
testpmd> set vf vlan tag (port_id) (vf_id) (on|off)
```

vlan set antispoof (for VF)

Set VLAN antispoof for a VF from the PF:

```
testpmd> set vf vlan antispoof (port_id) (vf_id) (on|off)
```

vlan set (strip|filter|qinq_strip|extend)

Set the VLAN strip/filter/QinQ strip/extend on for a port:

```
testpmd> vlan set (strip|filter|qinq_strip|extend) (on|off) (port_id)
```

vlan set tpid

Set the inner or outer VLAN TPID for packet filtering on a port:

```
testpmd> vlan set (inner|outer) tpid (value) (port_id)
```

Note: TPID value must be a 16-bit number (value <= 65536).

rx_vlan add

Add a VLAN ID, or all identifiers, to the set of VLAN identifiers filtered by port ID:

```
testpmd> rx_vlan add (vlan_id|all) (port_id)
```

Note: VLAN filter must be set on that port. VLAN ID < 4096. Depending on the NIC used, number of vlan_ids may be limited to the maximum entries in VFTA table. This is important if enabling all vlan_ids.

rx_vlan rm

Remove a VLAN ID, or all identifiers, from the set of VLAN identifiers filtered by port ID:

```
testpmd> rx_vlan rm (vlan_id|all) (port_id)
```

rx_vlan add (for VF)

Add a VLAN ID, to the set of VLAN identifiers filtered for VF(s) for port ID:

```
testpmd> rx_vlan add (vlan_id) port (port_id) vf (vf_mask)
```

rx_vlan rm (for VF)

Remove a VLAN ID, from the set of VLAN identifiers filtered for VF(s) for port ID:

```
testpmd> rx_vlan rm (vlan_id) port (port_id) vf (vf_mask)
```

tunnel_filter add

Add a tunnel filter on a port:

```
testpmd> tunnel_filter add (port_id) (outer_mac) (inner_mac) (ip_addr) \
         (inner_vlan) (vxlan|nvgre|ipingre|vxlan-gpe) (imac-ivlan|imac-ivlan-tenid|\
         imac-tenid|imac|omac-imac-tenid|oip|iip) (tenant_id) (queue_id)
```

The available information categories are:

- `vxlan`: Set tunnel type as VXLAN.
- `nvgre`: Set tunnel type as NVGRE.
- `ipingre`: Set tunnel type as IP-in-GRE.
- `vxlan-gpe`: Set tunnel type as VXLAN-GPE
- `imac-ivlan`: Set filter type as Inner MAC and VLAN.
- `imac-ivlan-tenid`: Set filter type as Inner MAC, VLAN and tenant ID.
- `imac-tenid`: Set filter type as Inner MAC and tenant ID.
- `imac`: Set filter type as Inner MAC.
- `omac-imac-tenid`: Set filter type as Outer MAC, Inner MAC and tenant ID.
- `oip`: Set filter type as Outer IP.
- `iip`: Set filter type as Inner IP.

Example:

```
testpmd> tunnel_filter add 0 68:05:CA:28:09:82 00:00:00:00:00:00 \
         192.168.2.2 0 ipingre oip 1 1

Set an IP-in-GRE tunnel on port 0, and the filter type is Outer IP.
```

tunnel_filter remove

Remove a tunnel filter on a port:

```
testpmd> tunnel_filter rm (port_id) (outer_mac) (inner_mac) (ip_addr) \
         (inner_vlan) (vxlan|nvgre|ipingre|vxlan-gpe) (imac-ivlan|imac-ivlan-tenid|\
         imac-tenid|imac|omac-imac-tenid|oip|iip) (tenant_id) (queue_id)
```

rx_vxlan_port add

Add an UDP port for VXLAN packet filter on a port:

```
testpmd> rx_vxlan_port add (udp_port) (port_id)
```

rx_vxlan_port remove

Remove an UDP port for VXLAN packet filter on a port:

```
testpmd> rx_vxlan_port rm (udp_port) (port_id)
```

tx_vlan set

Set hardware insertion of VLAN IDs in packets sent on a port:

```
testpmd> tx_vlan set (port_id) vlan_id[, vlan_id_outer]
```

For example, set a single VLAN ID (5) insertion on port 0:

```
tx_vlan set 0 5
```

Or, set double VLAN ID (inner: 2, outer: 3) insertion on port 1:

```
tx_vlan set 1 2 3
```

tx_vlan set pvid

Set port based hardware insertion of VLAN ID in packets sent on a port:

```
testpmd> tx_vlan set pvid (port_id) (vlan_id) (on|off)
```

tx_vlan reset

Disable hardware insertion of a VLAN header in packets sent on a port:

```
testpmd> tx_vlan reset (port_id)
```

csum set

Select hardware or software calculation of the checksum when transmitting a packet using the `csum` forwarding engine:

```
testpmd> csum set (ip|udp|tcp|sctp|outer-ip|outer-udp) (hw|sw) (port_id)
```

Where:

- `ip|udp|tcp|sctp` always relate to the inner layer.
- `outer-ip` relates to the outer IP layer (only for IPv4) in the case where the packet is recognized as a tunnel packet by the forwarding engine (vxlan, gre and ipip are supported). See also the `csum parse-tunnel` command.
- `outer-udp` relates to the outer UDP layer in the case where the packet is recognized as a tunnel packet by the forwarding engine (vxlan, vxlan-gpe are supported). See also the `csum parse-tunnel` command.

Note: Check the NIC Datasheet for hardware limits.

RSS queue region

Set RSS queue region span on a port:

```
testpmd> set port (port_id) queue-region region_id (value) \
             queue_start_index (value) queue_num (value)
```

Set flowtype mapping on a RSS queue region on a port:

```
testpmd> set port (port_id) queue-region region_id (value) flowtype (value)
```

where:

- For the flowtype(pctype) of packet,the specific index for each type has been defined in file i40e_type.h as enum i40e_filter_pctype.

Set user priority mapping on a RSS queue region on a port:

```
testpmd> set port (port_id) queue-region UP (value) region_id (value)
```

Flush all queue region related configuration on a port:

```
testpmd> set port (port_id) queue-region flush (on|off)
```

where:

- `on`: is just an enable function which server for other configuration, it is for all configuration about queue region from up layer, at first will only keep in DPDK software stored in driver, only after "flush on", it commit all configuration to HW.
- `"off`: is just clean all configuration about queue region just now, and restore all to DPDK i40e driver default config when start up.

Show all queue region related configuration info on a port:

```
testpmd> show port (port_id) queue-region
```

Note: Queue region only support on PF by now, so these command is only for configuration of queue region on PF port.

csum parse-tunnel

Define how tunneled packets should be handled by the csum forward engine:

```
testpmd> csum parse-tunnel (on|off) (tx_port_id)
```

If enabled, the csum forward engine will try to recognize supported tunnel headers (vxlan, gre, ipip).

If disabled, treat tunnel packets as non-tunneled packets (a inner header is handled as a packet payload).

Note: The port argument is the TX port like in the `csum set` command.

Example:

Consider a packet in packet like the following:

```
eth_out/ipv4_out/udp_out/vxlan/eth_in/ipv4_in/tcp_in
```

- If parse-tunnel is enabled, the `ip|udp|tcp|sctp` parameters of `csum set` command relate to the inner headers (here `ipv4_in` and `tcp_in`), and the `outer-ip|outer-udp` parameter relates to the outer headers (here `ipv4_out` and `udp_out`).
- **If parse-tunnel is disabled, the `ip|udp|tcp|sctp` parameters of `csum set`** command relate to the outer headers, here `ipv4_out` and `udp_out`.

csum show

Display tx checksum offload configuration:

```
testpmd> csum show (port_id)
```

tso set

Enable TCP Segmentation Offload (TSO) in the `csum` forwarding engine:

```
testpmd> tso set (segsize) (port_id)
```

Note: Check the NIC datasheet for hardware limits.

tso show

Display the status of TCP Segmentation Offload:

```
testpmd> tso show (port_id)
```

tunnel tso set

Set tso segment size of tunneled packets for a port in csum engine:

```
testpmd> tunnel_tso set (tso_segsz) (port_id)
```

tunnel tso show

Display the status of tunneled TCP Segmentation Offload for a port:

```
testpmd> tunnel_tso show (port_id)
```

set port - gro

Enable or disable GRO in `csum` forwarding engine:

```
testpmd> set port <port_id> gro on|off
```

If enabled, the csum forwarding engine will perform GRO on the TCP/IPv4 packets received from the given port.

If disabled, packets received from the given port won't be performed GRO. By default, GRO is disabled for all ports.

Note: When enable GRO for a port, TCP/IPv4 packets received from the port will be performed GRO. After GRO, all merged packets have bad checksums, since the GRO library doesn't re-calculate checksums for the merged packets. Therefore, if users want the merged packets to have correct checksums, please select HW IP checksum calculation and HW TCP checksum calculation for the port which the merged packets are transmitted to.

---

show port - gro

Display GRO configuration for a given port:

```
testpmd> show port <port_id> gro
```

set gro flush

Set the cycle to flush the GROed packets from reassembly tables:

```
testpmd> set gro flush <cycles>
```

When enable GRO, the csum forwarding engine performs GRO on received packets, and the GROed packets are stored in reassembly tables. Users can use this command to determine when the GROed packets are flushed from the reassembly tables.

The `cycles` is measured in GRO operation times. The csum forwarding engine flushes the GROed packets from the tables every `cycles` GRO operations.

By default, the value of `cycles` is 1, which means flush GROed packets from the reassembly tables as soon as one GRO operation finishes. The value of `cycles` should be in the range of 1 to `GRO_MAX_FLUSH_CYCLES`.

Please note that the large value of `cycles` may cause the poor TCP/IP stack performance. Because the GROed packets are delayed to arrive the stack, thus causing more duplicated ACKs and TCP retransmissions.

set port - gso

Toggle per-port GSO support in `csum` forwarding engine:

```
testpmd> set port <port_id> gso on|off
```

If enabled, the csum forwarding engine will perform GSO on supported IPv4 packets, transmitted on the given port.

If disabled, packets transmitted on the given port will not undergo GSO. By default, GSO is disabled for all ports.

Note: When GSO is enabled on a port, supported IPv4 packets transmitted on that port undergo GSO. Afterwards, the segmented packets are represented by multi-segment mbufs; however, the csum forwarding engine doesn't calculation of checksums for GSO'd segments in SW. As a result, if users want correct checksums in GSO segments, they should enable HW checksum calculation for GSO-enabled ports.

For example, HW checksum calculation for VxLAN GSO'd packets may be enabled by setting the following options in the csum forwarding engine:

testpmd> csum set outer_ip hw <port_id>

testpmd> csum set ip hw <port_id>

testpmd> csum set tcp hw <port_id>

UDP GSO is the same as IP fragmentation, which treats the UDP header as the payload and does not modify it during segmentation. That is, after UDP GSO, only the first output fragment has the original UDP header. Therefore, users need to enable HW IP checksum calculation and SW UDP checksum calculation for GSO-enabled ports, if they want correct checksums for UDP/IPv4 packets.

---

set gso segsz

Set the maximum GSO segment size (measured in bytes), which includes the packet header and the packet payload for GSO-enabled ports (global):

```
testpmd> set gso segsz <length>
```

show port - gso

Display the status of Generic Segmentation Offload for a given port:

```
testpmd> show port <port_id> gso
```

mac_addr add

Add an alternative MAC address to a port:

```
testpmd> mac_addr add (port_id) (XX:XX:XX:XX:XX:XX)
```

mac_addr remove

Remove a MAC address from a port:

```
testpmd> mac_addr remove (port_id) (XX:XX:XX:XX:XX:XX)
```

mcast_addr add

To add the multicast MAC address to/from the set of multicast addresses filtered by port:

```
testpmd> mcast_addr add (port_id) (mcast_addr)
```

mcast_addr remove

To remove the multicast MAC address to/from the set of multicast addresses filtered by port:

```
testpmd> mcast_addr remove (port_id) (mcast_addr)
```

mac_addr add (for VF)

Add an alternative MAC address for a VF to a port:

```
testpmd> mac_add add port (port_id) vf (vf_id) (XX:XX:XX:XX:XX:XX)
```

mac_addr set

Set the default MAC address for a port:

```
testpmd> mac_addr set (port_id) (XX:XX:XX:XX:XX:XX)
```

mac_addr set (for VF)

Set the MAC address for a VF from the PF:

```
testpmd> set vf mac addr (port_id) (vf_id) (XX:XX:XX:XX:XX:XX)
```

set eth-peer

Set the forwarding peer address for certain port:

```
testpmd> set eth-peer (port_id) (peer_addr)
```

This is equivalent to the `--eth-peer` command-line option.

set port-uta

Set the unicast hash filter(s) on/off for a port:

```
testpmd> set port (port_id) uta (XX:XX:XX:XX:XX:XX|all) (on|off)
```

set promisc

Set the promiscuous mode on for a port or for all ports. In promiscuous mode packets are not dropped if they aren't for the specified MAC address:

```
testpmd> set promisc (port_id|all) (on|off)
```

set allmulti

Set the allmulti mode for a port or for all ports:

```
testpmd> set allmulti (port_id|all) (on|off)
```

Same as the ifconfig (8) option. Controls how multicast packets are handled.

set promisc (for VF)

Set the unicast promiscuous mode for a VF from PF. It's supported by Intel i40e NICs now. In promiscuous mode packets are not dropped if they aren't for the specified MAC address:

```
testpmd> set vf promisc (port_id) (vf_id) (on|off)
```

set allmulticast (for VF)

Set the multicast promiscuous mode for a VF from PF. It's supported by Intel i40e NICs now. In promiscuous mode packets are not dropped if they aren't for the specified MAC address:

```
testpmd> set vf allmulti (port_id) (vf_id) (on|off)
```

set tx max bandwidth (for VF)

Set TX max absolute bandwidth (Mbps) for a VF from PF:

```
testpmd> set vf tx max-bandwidth (port_id) (vf_id) (max_bandwidth)
```

set tc tx min bandwidth (for VF)

Set all TCs' TX min relative bandwidth (%) for a VF from PF:

```
testpmd> set vf tc tx min-bandwidth (port_id) (vf_id) (bw1, bw2, ...)
```

set tc tx max bandwidth (for VF)

Set a TC's TX max absolute bandwidth (Mbps) for a VF from PF:

```
testpmd> set vf tc tx max-bandwidth (port_id) (vf_id) (tc_no) (max_bandwidth)
```

set tc strict link priority mode

Set some TCs' strict link priority mode on a physical port:

```
testpmd> set tx strict-link-priority (port_id) (tc_bitmap)
```

set tc tx min bandwidth

Set all TCs' TX min relative bandwidth (%) globally for all PF and VFs:

```
testpmd> set tc tx min-bandwidth (port_id) (bw1, bw2, ...)
```

set flow_ctrl rx

Set the link flow control parameter on a port:

```
testpmd> set flow_ctrl rx (on|off) tx (on|off) (high_water) (low_water) \
         (pause_time) (send_xon) mac_ctrl_frame_fwd (on|off) \
         autoneg (on|off) (port_id)
```

Where:

- `high_water` (integer): High threshold value to trigger XOFF.
- `low_water` (integer): Low threshold value to trigger XON.
- `pause_time` (integer): Pause quota in the Pause frame.
- `send_xon` (0/1): Send XON frame.
- `mac_ctrl_frame_fwd`: Enable receiving MAC control frames.
- `autoneg`: Change the auto-negotiation parameter.

set pfc_ctrl rx

Set the priority flow control parameter on a port:

```
testpmd> set pfc_ctrl rx (on|off) tx (on|off) (high_water) (low_water) \
         (pause_time) (priority) (port_id)
```

Where:

- `high_water` (integer): High threshold value.
- `low_water` (integer): Low threshold value.
- `pause_time` (integer): Pause quota in the Pause frame.
- `priority` (0-7): VLAN User Priority.

set stat_qmap

Set statistics mapping (qmapping 0..15) for RX/TX queue on port:

```
testpmd> set stat_qmap (tx|rx) (port_id) (queue_id) (qmapping)
```

For example, to set rx queue 2 on port 0 to mapping 5:

```
testpmd>set stat_qmap rx 0 2 5
```

set xstats-hide-zero

Set the option to hide zero values for xstats display:

```
testpmd> set xstats-hide-zero on|off
```

Note: By default, the zero values are displayed for xstats.

set port - rx/tx (for VF)

Set VF receive/transmit from a port:

```
testpmd> set port (port_id) vf (vf_id) (rx|tx) (on|off)
```

set port - mac address filter (for VF)

Add/Remove unicast or multicast MAC addr filter for a VF:

```
testpmd> set port (port_id) vf (vf_id) (mac_addr) \
         (exact-mac|exact-mac-vlan|hashmac|hashmac-vlan) (on|off)
```

set port - rx mode(for VF)

Set the VF receive mode of a port:

```
testpmd> set port (port_id) vf (vf_id) \
         rxmode (AUPE|ROPE|BAM|MPE) (on|off)
```

The available receive modes are:

- `AUPE`: Accepts untagged VLAN.
- `ROPE`: Accepts unicast hash.
- `BAM`: Accepts broadcast packets.
- `MPE`: Accepts all multicast packets.

set port - tx_rate (for Queue)

Set TX rate limitation for a queue on a port:

```
testpmd> set port (port_id) queue (queue_id) rate (rate_value)
```

set port - tx_rate (for VF)

Set TX rate limitation for queues in VF on a port:

```
testpmd> set port (port_id) vf (vf_id) rate (rate_value) queue_mask (queue_mask)
```

set port - mirror rule

Set pool or vlan type mirror rule for a port:

```
testpmd> set port (port_id) mirror-rule (rule_id) \
         (pool-mirror-up|pool-mirror-down|vlan-mirror) \
         (poolmask|vlanid[,vlanid]*) dst-pool (pool_id) (on|off)
```

Set link mirror rule for a port:

```
testpmd> set port (port_id) mirror-rule (rule_id) \
        (uplink-mirror|downlink-mirror) dst-pool (pool_id) (on|off)
```

For example to enable mirror traffic with vlan 0,1 to pool 0:

```
set port 0 mirror-rule 0 vlan-mirror 0,1 dst-pool 0 on
```

reset port - mirror rule

Reset a mirror rule for a port:

```
testpmd> reset port (port_id) mirror-rule (rule_id)
```

set flush_rx

Set the flush on RX streams before forwarding. The default is flush on. Mainly used with PCAP drivers to turn off the default behavior of flushing the first 512 packets on RX streams:

```
testpmd> set flush_rx off
```

set bypass mode

Set the bypass mode for the lowest port on bypass enabled NIC:

```
testpmd> set bypass mode (normal|bypass|isolate) (port_id)
```

set bypass event

Set the event required to initiate specified bypass mode for the lowest port on a bypass enabled:

```
testpmd> set bypass event (timeout|os_on|os_off|power_on|power_off) \
         mode (normal|bypass|isolate) (port_id)
```

Where:

- `timeout`: Enable bypass after watchdog timeout.
- `os_on`: Enable bypass when OS/board is powered on.
- `os_off`: Enable bypass when OS/board is powered off.
- `power_on`: Enable bypass when power supply is turned on.
- `power_off`: Enable bypass when power supply is turned off.

set bypass timeout

Set the bypass watchdog timeout to `n` seconds where 0 = instant:

```
testpmd> set bypass timeout (0|1.5|2|3|4|8|16|32)
```

show bypass config

Show the bypass configuration for a bypass enabled NIC using the lowest port on the NIC:

```
testpmd> show bypass config (port_id)
```

set link up

Set link up for a port:

```
testpmd> set link-up port (port id)
```

set link down

Set link down for a port:

```
testpmd> set link-down port (port id)
```

E-tag set

Enable E-tag insertion for a VF on a port:

```
testpmd> E-tag set insertion on port-tag-id (value) port (port_id) vf (vf_id)
```

Disable E-tag insertion for a VF on a port:

```
testpmd> E-tag set insertion off port (port_id) vf (vf_id)
```

Enable/disable E-tag stripping on a port:

```
testpmd> E-tag set stripping (on|off) port (port_id)
```

Enable/disable E-tag based forwarding on a port:

```
testpmd> E-tag set forwarding (on|off) port (port_id)
```

Add an E-tag forwarding filter on a port:

```
testpmd> E-tag set filter add e-tag-id (value) dst-pool (pool_id) port (port_id)
```

Delete an E-tag forwarding filter on a port::
testpmd> E-tag set filter del e-tag-id (value) port (port_id)

ddp add

Load a dynamic device personalization (DDP) profile and store backup profile:

```
testpmd> ddp add (port_id) (profile_path[,backup_profile_path])
```

ddp del

Delete a dynamic device personalization profile and restore backup profile:

```
testpmd> ddp del (port_id) (backup_profile_path)
```

ptype mapping

List all items from the ptype mapping table:

```
testpmd> ptype mapping get (port_id) (valid_only)
```

Where:

- `valid_only`: A flag indicates if only list valid items(=1) or all itemss(=0).

Replace a specific or a group of software defined ptype with a new one:

```
testpmd> ptype mapping replace  (port_id) (target) (mask) (pkt_type)
```

where:

- `target`: A specific software ptype or a mask to represent a group of software ptypes.

- `mask`: A flag indicate if "target" is a specific software ptype(=0) or a ptype mask(=1).
- `pkt_type`: The new software ptype to replace the old ones.

Update hardware defined ptype to software defined packet type mapping table:

```
testpmd> ptype mapping update (port_id) (hw_ptype) (sw_ptype)
```

where:

- `hw_ptype`: hardware ptype as the index of the ptype mapping table.
- `sw_ptype`: software ptype as the value of the ptype mapping table.

Reset ptype mapping table:

```
testpmd> ptype mapping reset (port_id)
```

config per port Rx offloading

Enable or disable a per port Rx offloading on all Rx queues of a port:

```
testpmd> port config (port_id) rx_offload (offloading) on|off
```

- **`offloading`: can be any of these offloading capability:**
  vlan_strip, ipv4_cksum, udp_cksum, tcp_cksum, tcp_lro, qinq_strip, outer_ipv4_cksum, macsec_strip, header_split, vlan_filter, vlan_extend, jumbo_frame, scatter, timestamp, security, keep_crc, rss_hash

This command should be run when the port is stopped, or else it will fail.

config per queue Rx offloading

Enable or disable a per queue Rx offloading only on a specific Rx queue:

```
testpmd> port (port_id) rxq (queue_id) rx_offload (offloading) on|off
```

- **`offloading`: can be any of these offloading capability:**
  vlan_strip, ipv4_cksum, udp_cksum, tcp_cksum, tcp_lro, qinq_strip, outer_ipv4_cksum, macsec_strip, header_split, vlan_filter, vlan_extend, jumbo_frame, scatter, timestamp, security, keep_crc

This command should be run when the port is stopped, or else it will fail.

config per port Tx offloading

Enable or disable a per port Tx offloading on all Tx queues of a port:

```
testpmd> port config (port_id) tx_offload (offloading) on|off
```

- **`offloading`: can be any of these offloading capability:**
  vlan_insert, ipv4_cksum, udp_cksum, tcp_cksum, sctp_cksum, tcp_tso, udp_tso, outer_ipv4_cksum, qinq_insert, vxlan_tnl_tso, gre_tnl_tso, ipip_tnl_tso, geneve_tnl_tso, macsec_insert, mt_lockfree, multi_segs, mbuf_fast_free, security

This command should be run when the port is stopped, or else it will fail.

config per queue Tx offloading

Enable or disable a per queue Tx offloading only on a specific Tx queue:

```
testpmd> port (port_id) txq (queue_id) tx_offload (offloading) on|off
```

- **`offloading`: can be any of these offloading capability:**
  vlan_insert, ipv4_cksum, udp_cksum, tcp_cksum, sctp_cksum, tcp_tso, udp_tso, outer_ipv4_cksum, qinq_insert, vxlan_tnl_tso, gre_tnl_tso, ipip_tnl_tso, geneve_tnl_tso, macsec_insert, mt_lockfree, multi_segs, mbuf_fast_free, security

This command should be run when the port is stopped, or else it will fail.

Config VXLAN Encap outer layers

Configure the outer layer to encapsulate a packet inside a VXLAN tunnel:

```
set vxlan ip-version (ipv4|ipv6) vni (vni) udp-src (udp-src) \
udp-dst (udp-dst) ip-src (ip-src) ip-dst (ip-dst) eth-src (eth-src) \
eth-dst (eth-dst)

set vxlan-with-vlan ip-version (ipv4|ipv6) vni (vni) udp-src (udp-src) \
udp-dst (udp-dst) ip-src (ip-src) ip-dst (ip-dst) vlan-tci (vlan-tci) \
eth-src (eth-src) eth-dst (eth-dst)

set vxlan-tos-ttl ip-version (ipv4|ipv6) vni (vni) udp-src (udp-src) \
udp-dst (udp-dst) ip-tos (ip-tos) ip-ttl (ip-ttl) ip-src (ip-src) \
ip-dst (ip-dst) eth-src (eth-src) eth-dst (eth-dst)
```

These commands will set an internal configuration inside testpmd, any following flow rule using the action vxlan_encap will use the last configuration set. To have a different encapsulation header, one of those commands must be called before the flow rule creation.

Config NVGRE Encap outer layers

Configure the outer layer to encapsulate a packet inside a NVGRE tunnel:

```
set nvgre ip-version (ipv4|ipv6) tni (tni) ip-src (ip-src) ip-dst (ip-dst) \
       eth-src (eth-src) eth-dst (eth-dst)
set nvgre-with-vlan ip-version (ipv4|ipv6) tni (tni) ip-src (ip-src) \
       ip-dst (ip-dst) vlan-tci (vlan-tci) eth-src (eth-src) eth-dst (eth-dst)
```

These commands will set an internal configuration inside testpmd, any following flow rule using the action nvgre_encap will use the last configuration set. To have a different encapsulation header, one of those commands must be called before the flow rule creation.

Config L2 Encap

Configure the l2 to be used when encapsulating a packet with L2:

```
set l2_encap ip-version (ipv4|ipv6) eth-src (eth-src) eth-dst (eth-dst)
set l2_encap-with-vlan ip-version (ipv4|ipv6) vlan-tci (vlan-tci) \
       eth-src (eth-src) eth-dst (eth-dst)
```

Those commands will set an internal configuration inside testpmd, any following flow rule using the action l2_encap will use the last configuration set. To have a different encapsulation header, one of those commands must be called before the flow rule creation.

Config L2 Decap

Configure the l2 to be removed when decapsulating a packet with L2:

```
set l2_decap ip-version (ipv4|ipv6)
set l2_decap-with-vlan ip-version (ipv4|ipv6)
```

Those commands will set an internal configuration inside testpmd, any following flow rule using the action l2_decap will use the last configuration set. To have a different encapsulation header, one of those commands must be called before the flow rule creation.

Config MPLSoGRE Encap outer layers

Configure the outer layer to encapsulate a packet inside a MPLSoGRE tunnel:

```
set mplsogre_encap ip-version (ipv4|ipv6) label (label) \
       ip-src (ip-src) ip-dst (ip-dst) eth-src (eth-src) eth-dst (eth-dst)
set mplsogre_encap-with-vlan ip-version (ipv4|ipv6) label (label) \
       ip-src (ip-src) ip-dst (ip-dst) vlan-tci (vlan-tci) \
       eth-src (eth-src) eth-dst (eth-dst)
```

These commands will set an internal configuration inside testpmd, any following flow rule using the action mplsogre_encap will use the last configuration set. To have a different encapsulation header, one of those commands must be called before the flow rule creation.

Config MPLSoGRE Decap outer layers

Configure the outer layer to decapsulate MPLSoGRE packet:

```
set mplsogre_decap ip-version (ipv4|ipv6)
set mplsogre_decap-with-vlan ip-version (ipv4|ipv6)
```

These commands will set an internal configuration inside testpmd, any following flow rule using the action mplsogre_decap will use the last configuration set. To have a different decapsulation header, one of those commands must be called before the flow rule creation.

Config MPLSoUDP Encap outer layers

Configure the outer layer to encapsulate a packet inside a MPLSoUDP tunnel:

```
set mplsoudp_encap ip-version (ipv4|ipv6) label (label) udp-src (udp-src) \
       udp-dst (udp-dst) ip-src (ip-src) ip-dst (ip-dst) \
       eth-src (eth-src) eth-dst (eth-dst)
set mplsoudp_encap-with-vlan ip-version (ipv4|ipv6) label (label) \
       udp-src (udp-src) udp-dst (udp-dst) ip-src (ip-src) ip-dst (ip-dst) \
       vlan-tci (vlan-tci) eth-src (eth-src) eth-dst (eth-dst)
```

These commands will set an internal configuration inside testpmd, any following flow rule using the action mplsoudp_encap will use the last configuration set. To have a different encapsulation header, one of those commands must be called before the flow rule creation.

Config MPLSoUDP Decap outer layers

Configure the outer layer to decapsulate MPLSoUDP packet:

```
set mplsoudp_decap ip-version (ipv4|ipv6)
set mplsoudp_decap-with-vlan ip-version (ipv4|ipv6)
```

These commands will set an internal configuration inside testpmd, any following flow rule using the action mplsoudp_decap will use the last configuration set. To have a different decapsulation header, one of those commands must be called before the flow rule creation.

Config Raw Encapsulation

Configure the raw data to be used when encapsulating a packet by rte_flow_action_raw_encap:

```
set raw_encap {index} {item} [/ {item} [...]] / end_set
```

There are multiple global buffers for `raw_encap`, this command will set one internal buffer index by `{index}`. If there is no `{index}` specified:

```
set raw_encap {item} [/ {item} [...]] / end_set
```

the default index `0` is used. In order to use different encapsulating header, `index` must be specified during the flow rule creation:

```
testpmd> flow create 0 egress pattern eth / ipv4 / end actions
       raw_encap index 2 / end
```

Otherwise the default index `0` is used.

Config Raw Decapsulation

Configure the raw data to be used when decapsulating a packet by rte_flow_action_raw_decap:

```
set raw_decap {index} {item} [/ {item} [...]] / end_set
```

There are multiple global buffers for `raw_decap`, this command will set one internal buffer index by `{index}`. If there is no `{index}` specified:

```
set raw_decap {item} [/ {item} [...]] / end_set
```

the default index `0` is used. In order to use different decapsulating header, `index` must be specified during the flow rule creation:

```
testpmd> flow create 0 egress pattern eth / ipv4 / end actions
         raw_encap index 3 / end
```

Otherwise the default index `0` is used.

8.4.6 Port Functions

The following sections show functions for configuring ports.

Note: Port configuration changes only become active when forwarding is started/restarted.

port attach

Attach a port specified by pci address or virtual device args:

```
testpmd> port attach (identifier)
```

To attach a new pci device, the device should be recognized by kernel first. Then it should be moved under DPDK management. Finally the port can be attached to testpmd.

For example, to move a pci device using ixgbe under DPDK management:

```
# Check the status of the available devices.
./usertools/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
============================================
<none>

Network devices using kernel driver
===================================
0000:0a:00.0 '82599ES 10-Gigabit' if=eth2 drv=ixgbe unused=


# Bind the device to igb_uio.
sudo ./usertools/dpdk-devbind.py -b igb_uio 0000:0a:00.0


# Recheck the status of the devices.
./usertools/dpdk-devbind.py --status
```

(continues on next page)

(continued from previous page)

```
Network devices using DPDK-compatible driver
============================================
0000:0a:00.0 '82599ES 10-Gigabit' drv=igb_uio unused=
```

To attach a port created by virtual device, above steps are not needed.

For example, to attach a port whose pci address is 0000:0a:00.0.

```
testpmd> port attach 0000:0a:00.0
Attaching a new port...
EAL: PCI device 0000:0a:00.0 on NUMA socket -1
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x7f83bfa00000
EAL:   PCI memory mapped at 0x7f83bfa80000
PMD: eth_ixgbe_dev_init(): MAC: 2, PHY: 18, SFP+: 5
PMD: eth_ixgbe_dev_init(): port 0 vendorID=0x8086 deviceID=0x10fb
Port 0 is attached. Now total ports is 1
Done
```

For example, to attach a port created by pcap PMD.

```
testpmd> port attach net_pcap0
Attaching a new port...
PMD: Initializing pmd_pcap for net_pcap0
PMD: Creating pcap-backed ethdev on numa socket 0
Port 0 is attached. Now total ports is 1
Done
```

In this case, identifier is `net_pcap0`. This identifier format is the same as `--vdev` format of DPDK applications.

For example, to re-attach a bonded port which has been previously detached, the mode and slave parameters must be given.

```
testpmd> port attach net_bond_0,mode=0,slave=1
Attaching a new port...
EAL: Initializing pmd_bond for net_bond_0
EAL: Create bonded device net_bond_0 on port 0 in mode 0 on socket 0.
Port 0 is attached. Now total ports is 1
Done
```

port detach

Detach a specific port:

```
testpmd> port detach (port_id)
```

Before detaching a port, the port should be stopped and closed.

For example, to detach a pci device port 0.

```
testpmd> port stop 0
Stopping ports...
Done
testpmd> port close 0
Closing ports...
Done
```

(continues on next page)

(continued from previous page)

```
testpmd> port detach 0
Detaching a port...
EAL: PCI device 0000:0a:00.0 on NUMA socket -1
EAL:   remove driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory unmapped at 0x7f83bfa00000
EAL:   PCI memory unmapped at 0x7f83bfa80000
Done
```

For example, to detach a virtual device port 0.

```
testpmd> port stop 0
Stopping ports...
Done
testpmd> port close 0
Closing ports...
Done

testpmd> port detach 0
Detaching a port...
PMD: Closing pcap ethdev on numa socket 0
Port 'net_pcap0' is detached. Now total ports is 0
Done
```

To remove a pci device completely from the system, first detach the port from testpmd. Then the device should be moved under kernel management. Finally the device can be removed using kernel pci hotplug functionality.

For example, to move a pci device under kernel management:

```
sudo ./usertools/dpdk-devbind.py -b ixgbe 0000:0a:00.0

./usertools/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
============================================
<none>

Network devices using kernel driver
===================================
0000:0a:00.0 '82599ES 10-Gigabit' if=eth2 drv=ixgbe unused=igb_uio
```

To remove a port created by a virtual device, above steps are not needed.

port start

Start all ports or a specific port:

```
testpmd> port start (port_id|all)
```

port stop

Stop all ports or a specific port:

```
testpmd> port stop (port_id|all)
```

port close

Close all ports or a specific port:

```
testpmd> port close (port_id|all)
```

port reset

Reset all ports or a specific port:

```
testpmd> port reset (port_id|all)
```

User should stop port(s) before resetting and (re-)start after reset.

port config - queue ring size

Configure a rx/tx queue ring size:

```
testpmd> port (port_id) (rxq|txq) (queue_id) ring_size (value)
```

Only take effect after command that (re-)start the port or command that setup specific queue.

port start/stop queue

Start/stop a rx/tx queue on a specific port:

```
testpmd> port (port_id) (rxq|txq) (queue_id) (start|stop)
```

port config - queue deferred start

Switch on/off deferred start of a specific port queue:

```
testpmd> port (port_id) (rxq|txq) (queue_id) deferred_start (on|off)
```

port setup queue

Setup a rx/tx queue on a specific port:

```
testpmd> port (port_id) (rxq|txq) (queue_id) setup
```

Only take effect when port is started.

port config - speed

Set the speed and duplex mode for all ports or a specific port:

```
testpmd> port config (port_id|all) speed␣
↪(10|100|1000|10000|25000|40000|50000|100000|200000|auto) \
        duplex (half|full|auto)
```

port config - queues/descriptors

Set number of queues/descriptors for rxq, txq, rxd and txd:

```
testpmd> port config all (rxq|txq|rxd|txd) (value)
```

This is equivalent to the `--rxq`, `--txq`, `--rxd` and `--txd` command-line options.

port config - max-pkt-len

Set the maximum packet length:

```
testpmd> port config all max-pkt-len (value)
```

This is equivalent to the `--max-pkt-len` command-line option.

port config - max-lro-pkt-size

Set the maximum LRO aggregated packet size:

```
testpmd> port config all max-lro-pkt-size (value)
```

This is equivalent to the `--max-lro-pkt-size` command-line option.

port config - Drop Packets

Enable or disable packet drop on all RX queues of all ports when no receive buffers available:

```
testpmd> port config all drop-en (on|off)
```

Packet dropping when no receive buffers available is off by default.

The `on` option is equivalent to the `--enable-drop-en` command-line option.

port config - RSS

Set the RSS (Receive Side Scaling) mode on or off:

```
testpmd> port config all rss␣
↪(all|default|eth|vlan|ip|tcp|udp|sctp|ether|port|vxlan|geneve|nvgre|vxlan-
↪gpe|l2tpv3|esp|ah|pfcp|none)
```

RSS is on by default.

The `all` option is equivalent to eth|vlan|ip|tcp|udp|sctp|ether|l2tpv3|esp|ah|pfcp.

The `default` option enables all supported RSS types reported by device info.

The `none` option is equivalent to the `--disable-rss` command-line option.

port config - RSS Reta

Set the RSS (Receive Side Scaling) redirection table:

```
testpmd> port config all rss reta (hash,queue)[,(hash,queue)]
```

port config - DCB

Set the DCB mode for an individual port:

```
testpmd> port config (port_id) dcb vt (on|off) (traffic_class) pfc (on|off)
```

The traffic class should be 4 or 8.

port config - Burst

Set the number of packets per burst:

```
testpmd> port config all burst (value)
```

This is equivalent to the `--burst` command-line option.

port config - Threshold

Set thresholds for TX/RX queues:

```
testpmd> port config all (threshold) (value)
```

Where the threshold type can be:

- `txpt:` Set the prefetch threshold register of the TX rings, 0 <= value <= 255.
- `txht:` Set the host threshold register of the TX rings, 0 <= value <= 255.
- `txwt:` Set the write-back threshold register of the TX rings, 0 <= value <= 255.
- `rxpt:` Set the prefetch threshold register of the RX rings, 0 <= value <= 255.
- `rxht:` Set the host threshold register of the RX rings, 0 <= value <= 255.
- `rxwt:` Set the write-back threshold register of the RX rings, 0 <= value <= 255.
- `txfreet:` Set the transmit free threshold of the TX rings, 0 <= value <= txd.
- `rxfreet:` Set the transmit free threshold of the RX rings, 0 <= value <= rxd.
- `txrst:` Set the transmit RS bit threshold of TX rings, 0 <= value <= txd.

These threshold options are also available from the command-line.

port config - E-tag

Set the value of ether-type for E-tag:

```
testpmd> port config (port_id|all) l2-tunnel E-tag ether-type (value)
```

Enable/disable the E-tag support:

```
testpmd> port config (port_id|all) l2-tunnel E-tag (enable|disable)
```

port config pctype mapping

Reset pctype mapping table:

```
testpmd> port config (port_id) pctype mapping reset
```

Update hardware defined pctype to software defined flow type mapping table:

```
testpmd> port config (port_id) pctype mapping update (pctype_id_0[,pctype_id_1]*) (flow_type_
↪id)
```

where:

- `pctype_id_x`: hardware pctype id as index of bit in bitmask value of the pctype mapping table.
- `flow_type_id`: software flow type id as the index of the pctype mapping table.

port config input set

Config RSS/FDIR/FDIR flexible payload input set for some pctype:

```
testpmd> port config (port_id) pctype (pctype_id) \
         (hash_inset|fdir_inset|fdir_flx_inset) \
         (get|set|clear) field (field_idx)
```

Clear RSS/FDIR/FDIR flexible payload input set for some pctype:

```
testpmd> port config (port_id) pctype (pctype_id) \
         (hash_inset|fdir_inset|fdir_flx_inset) clear all
```

where:

- `pctype_id`: hardware packet classification types.
- `field_idx`: hardware field index.

port config udp_tunnel_port

Add/remove UDP tunnel port for VXLAN/GENEVE tunneling protocols:

```
testpmd> port config (port_id) udp_tunnel_port add|rm vxlan|geneve|vxlan-gpe (udp_port)
```

port config tx_metadata

Set Tx metadata value per port. testpmd will add this value to any Tx packet sent from this port:

```
testpmd> port config (port_id) tx_metadata (value)
```

port config dynf

Set/clear dynamic flag per port. testpmd will register this flag in the mbuf (same registration for both Tx and Rx). Then set/clear this flag for each Tx packet sent from this port. The set bit only works for Tx packet:

```
testpmd> port config (port_id) dynf (name) (set|clear)
```

port config mtu

To configure MTU(Maximum Transmission Unit) on devices using testpmd:

```
testpmd> port config mtu (port_id) (value)
```

port config rss hash key

To configure the RSS hash key used to compute the RSS hash of input [IP] packets received on port:

```
testpmd> port config <port_id> rss-hash-key (ipv4|ipv4-frag|\
                  ipv4-tcp|ipv4-udp|ipv4-sctp|ipv4-other|\
                  ipv6|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp|\
                  ipv6-other|l2-payload|ipv6-ex|ipv6-tcp-ex|\
                  ipv6-udp-ex <string of hex digits \
                  (variable length, NIC dependent)>)
```

8.4.7 Device Functions

The following sections show functions for device operations.

device detach

Detach a device specified by pci address or virtual device args:

```
testpmd> device detach (identifier)
```

Before detaching a device associated with ports, the ports should be stopped and closed.

For example, to detach a pci device whose address is 0002:03:00.0.

```
testpmd> device detach 0002:03:00.0
Removing a device...
Port 1 is now closed
EAL: Releasing pci mapped resource for 0002:03:00.0
EAL: Calling pci_unmap_resource for 0002:03:00.0 at 0x218a050000
EAL: Calling pci_unmap_resource for 0002:03:00.0 at 0x218c050000
Device 0002:03:00.0 is detached
Now total ports is 1
```

For example, to detach a port created by pcap PMD.

```
testpmd> device detach net_pcap0
Removing a device...
Port 0 is now closed
Device net_pcap0 is detached
Now total ports is 0
Done
```

In this case, identifier is `net_pcap0`. This identifier format is the same as `--vdev` format of DPDK applications.

8.4.8 Link Bonding Functions

The Link Bonding functions make it possible to dynamically create and manage link bonding devices from within testpmd interactive prompt.

create bonded device

Create a new bonding device:

```
testpmd> create bonded device (mode) (socket)
```

For example, to create a bonded device in mode 1 on socket 0:

```
testpmd> create bonded device 1 0
created new bonded device (port X)
```

add bonding slave

Adds Ethernet device to a Link Bonding device:

```
testpmd> add bonding slave (slave id) (port id)
```

For example, to add Ethernet device (port 6) to a Link Bonding device (port 10):

```
testpmd> add bonding slave 6 10
```

remove bonding slave

Removes an Ethernet slave device from a Link Bonding device:

```
testpmd> remove bonding slave (slave id) (port id)
```

For example, to remove Ethernet slave device (port 6) to a Link Bonding device (port 10):

```
testpmd> remove bonding slave 6 10
```

set bonding mode

Set the Link Bonding mode of a Link Bonding device:

```
testpmd> set bonding mode (value) (port id)
```

For example, to set the bonding mode of a Link Bonding device (port 10) to broadcast (mode 3):

```
testpmd> set bonding mode 3 10
```

set bonding primary

Set an Ethernet slave device as the primary device on a Link Bonding device:

```
testpmd> set bonding primary (slave id) (port id)
```

For example, to set the Ethernet slave device (port 6) as the primary port of a Link Bonding device (port 10):

```
testpmd> set bonding primary 6 10
```

set bonding mac

Set the MAC address of a Link Bonding device:

```
testpmd> set bonding mac (port id) (mac)
```

For example, to set the MAC address of a Link Bonding device (port 10) to 00:00:00:00:00:01:

```
testpmd> set bonding mac 10 00:00:00:00:00:01
```

set bonding xmit_balance_policy

Set the transmission policy for a Link Bonding device when it is in Balance XOR mode:

```
testpmd> set bonding xmit_balance_policy (port_id) (l2|l23|l34)
```

For example, set a Link Bonding device (port 10) to use a balance policy of layer 3+4 (IP addresses & UDP ports):

```
testpmd> set bonding xmit_balance_policy 10 l34
```

set bonding mon_period

Set the link status monitoring polling period in milliseconds for a bonding device.

This adds support for PMD slave devices which do not support link status interrupts. When the mon_period is set to a value greater than 0 then all PMD's which do not support link status ISR will be queried every polling interval to check if their link status has changed:

```
testpmd> set bonding mon_period (port_id) (value)
```

For example, to set the link status monitoring polling period of bonded device (port 5) to 150ms:

```
testpmd> set bonding mon_period 5 150
```

set bonding lacp dedicated_queue

Enable dedicated tx/rx queues on bonding devices slaves to handle LACP control plane traffic when in mode 4 (link-aggregation-802.3ad):

```
testpmd> set bonding lacp dedicated_queues (port_id) (enable|disable)
```

set bonding agg_mode

Enable one of the specific aggregators mode when in mode 4 (link-aggregation-802.3ad):

```
testpmd> set bonding agg_mode (port_id) (bandwidth|count|stable)
```

show bonding config

Show the current configuration of a Link Bonding device:

```
testpmd> show bonding config (port id)
```

For example, to show the configuration a Link Bonding device (port 9) with 3 slave devices (1, 3, 4) in balance mode with a transmission policy of layer 2+3:

```
testpmd> show bonding config 9
     Bonding mode: 2
     Balance Xmit Policy: BALANCE_XMIT_POLICY_LAYER23
     Slaves (3): [1 3 4]
     Active Slaves (3): [1 3 4]
     Primary: [3]
```

8.4.9 Register Functions

The Register Functions can be used to read from and write to registers on the network card referenced by a port number. This is mainly useful for debugging purposes. Reference should be made to the appropriate datasheet for the network card for details on the register addresses and fields that can be accessed.

read reg

Display the value of a port register:

```
testpmd> read reg (port_id) (address)
```

For example, to examine the Flow Director control register (FDIRCTL, 0x0000EE000) on an Intel 82599 10 GbE Controller:

```
testpmd> read reg 0 0xEE00
port 0 PCI register at offset 0xEE00: 0x4A060029 (1241907241)
```

read regfield

Display a port register bit field:

```
testpmd> read regfield (port_id) (address) (bit_x) (bit_y)
```

For example, reading the lowest two bits from the register in the example above:

```
testpmd> read regfield 0 0xEE00 0 1
port 0 PCI register at offset 0xEE00: bits[0, 1]=0x1 (1)
```

read regbit

Display a single port register bit:

```
testpmd> read regbit (port_id) (address) (bit_x)
```

For example, reading the lowest bit from the register in the example above:

```
testpmd> read regbit 0 0xEE00 0
port 0 PCI register at offset 0xEE00: bit 0=1
```

write reg

Set the value of a port register:

```
testpmd> write reg (port_id) (address) (value)
```

For example, to clear a register:

```
testpmd> write reg 0 0xEE00 0x0
port 0 PCI register at offset 0xEE00: 0x00000000 (0)
```

write regfield

Set bit field of a port register:

```
testpmd> write regfield (port_id) (address) (bit_x) (bit_y) (value)
```

For example, writing to the register cleared in the example above:

```
testpmd> write regfield 0 0xEE00 0 1 2
port 0 PCI register at offset 0xEE00: 0x00000002 (2)
```

write regbit

Set single bit value of a port register:

```
testpmd> write regbit (port_id) (address) (bit_x) (value)
```

For example, to set the high bit in the register from the example above:

```
testpmd> write regbit 0 0xEE00 31 1
port 0 PCI register at offset 0xEE00: 0x8000000A (2147483658)
```

8.4.10 Traffic Metering and Policing

The following section shows functions for configuring traffic metering and policing on the ethernet device through the use of generic ethdev API.

show port traffic management capability

Show traffic metering and policing capability of the port:

```
testpmd> show port meter cap (port_id)
```

add port meter profile (srTCM rfc2967)

Add meter profile (srTCM rfc2697) to the ethernet device:

```
testpmd> add port meter profile srtcm_rfc2697 (port_id) (profile_id) \
(cir) (cbs) (ebs)
```

where:

- `profile_id`: ID for the meter profile.
- `cir`: Committed Information Rate (CIR) (bytes/second).
- `cbs`: Committed Burst Size (CBS) (bytes).
- `ebs`: Excess Burst Size (EBS) (bytes).

add port meter profile (trTCM rfc2968)

Add meter profile (srTCM rfc2698) to the ethernet device:

```
testpmd> add port meter profile trtcm_rfc2698 (port_id) (profile_id) \
(cir) (pir) (cbs) (pbs)
```

where:

- `profile_id`: ID for the meter profile.
- `cir`: Committed information rate (bytes/second).
- `pir`: Peak information rate (bytes/second).

- `cbs`: Committed burst size (bytes).
- `pbs`: Peak burst size (bytes).

add port meter profile (trTCM rfc4115)

Add meter profile (trTCM rfc4115) to the ethernet device:

```
testpmd> add port meter profile trtcm_rfc4115 (port_id) (profile_id) \
(cir) (eir) (cbs) (ebs)
```

where:

- `profile_id`: ID for the meter profile.
- `cir`: Committed information rate (bytes/second).
- `eir`: Excess information rate (bytes/second).
- `cbs`: Committed burst size (bytes).
- `ebs`: Excess burst size (bytes).

delete port meter profile

Delete meter profile from the ethernet device:

```
testpmd> del port meter profile (port_id) (profile_id)
```

create port meter

Create new meter object for the ethernet device:

```
testpmd> create port meter (port_id) (mtr_id) (profile_id) \
(meter_enable) (g_action) (y_action) (r_action) (stats_mask) (shared) \
(use_pre_meter_color) [(dscp_tbl_entry0) (dscp_tbl_entry1)...\
(dscp_tbl_entry63)]
```

where:

- `mtr_id`: meter object ID.
- `profile_id`: ID for the meter profile.
- `meter_enable`: When this parameter has a non-zero value, the meter object gets enabled at the time of creation, otherwise remains disabled.
- `g_action`: Policer action for the packet with green color.
- `y_action`: Policer action for the packet with yellow color.
- `r_action`: Policer action for the packet with red color.
- `stats_mask`: Mask of statistics counter types to be enabled for the meter object.
- `shared`: When this parameter has a non-zero value, the meter object is shared by multiple flows. Otherwise, meter object is used by single flow.

- `use_pre_meter_color`: When this parameter has a non-zero value, the input color for the current meter object is determined by the latest meter object in the same flow. Otherwise, the current meter object uses the *dscp_table* to determine the input color.
- `dscp_tbl_entryx`: DSCP table entry x providing meter providing input color, 0 <= x <= 63.

enable port meter

Enable meter for the ethernet device:

```
testpmd> enable port meter (port_id) (mtr_id)
```

disable port meter

Disable meter for the ethernet device:

```
testpmd> disable port meter (port_id) (mtr_id)
```

delete port meter

Delete meter for the ethernet device:

```
testpmd> del port meter (port_id) (mtr_id)
```

Set port meter profile

Set meter profile for the ethernet device:

```
testpmd> set port meter profile (port_id) (mtr_id) (profile_id)
```

set port meter dscp table

Set meter dscp table for the ethernet device:

```
testpmd> set port meter dscp table (port_id) (mtr_id) [(dscp_tbl_entry0) \
(dscp_tbl_entry1)...(dscp_tbl_entry63)]
```

set port meter policer action

Set meter policer action for the ethernet device:

```
testpmd> set port meter policer action (port_id) (mtr_id) (action_mask) \
(action0) [(action1) (action1)]
```

where:

- `action_mask`: Bit mask indicating which policer actions need to be updated. One or more policer actions can be updated in a single function invocation. To update the policer action associated with color C, bit (1 << C) needs to be set in *action_mask* and element at position C in the *actions* array needs to be valid.
- `actionx`: Policer action for the color x, RTE_MTR_GREEN <= x < RTE_MTR_COLORS

set port meter stats mask

Set meter stats mask for the ethernet device:

```
testpmd> set port meter stats mask (port_id) (mtr_id) (stats_mask)
```

where:

- `stats_mask`: Bit mask indicating statistics counter types to be enabled.

show port meter stats

Show meter stats of the ethernet device:

```
testpmd> show port meter stats (port_id) (mtr_id) (clear)
```

where:

- `clear`: Flag that indicates whether the statistics counters should be cleared (i.e. set to zero) immediately after they have been read or not.

8.4.11 Traffic Management

The following section shows functions for configuring traffic management on the ethernet device through the use of generic TM API.

show port traffic management capability

Show traffic management capability of the port:

```
testpmd> show port tm cap (port_id)
```

show port traffic management capability (hierarchy level)

Show traffic management hierarchy level capability of the port:

```
testpmd> show port tm level cap (port_id) (level_id)
```

show port traffic management capability (hierarchy node level)

Show the traffic management hierarchy node capability of the port:

```
testpmd> show port tm node cap (port_id) (node_id)
```

show port traffic management hierarchy node type

Show the port traffic management hierarchy node type:

```
testpmd> show port tm node type (port_id) (node_id)
```

show port traffic management hierarchy node stats

Show the port traffic management hierarchy node statistics:

```
testpmd> show port tm node stats (port_id) (node_id) (clear)
```

where:

- `clear`: When this parameter has a non-zero value, the statistics counters are cleared (i.e. set to zero) immediately after they have been read, otherwise the statistics counters are left untouched.

Add port traffic management private shaper profile

Add the port traffic management private shaper profile:

```
testpmd> add port tm node shaper profile (port_id) (shaper_profile_id) \
(cmit_tb_rate) (cmit_tb_size) (peak_tb_rate) (peak_tb_size) \
(packet_length_adjust)
```

where:

- `shaper_profile id`: Shaper profile ID for the new profile.
- `cmit_tb_rate`: Committed token bucket rate (bytes per second).
- `cmit_tb_size`: Committed token bucket size (bytes).
- `peak_tb_rate`: Peak token bucket rate (bytes per second).
- `peak_tb_size`: Peak token bucket size (bytes).
- `packet_length_adjust`: The value (bytes) to be added to the length of each packet for the purpose of shaping. This parameter value can be used to correct the packet length with the framing overhead bytes that are consumed on the wire.

Delete port traffic management private shaper profile

Delete the port traffic management private shaper:

```
testpmd> del port tm node shaper profile (port_id) (shaper_profile_id)
```

where:

- `shaper_profile id`: Shaper profile ID that needs to be deleted.

Add port traffic management shared shaper

Create the port traffic management shared shaper:

```
testpmd> add port tm node shared shaper (port_id) (shared_shaper_id) \
(shaper_profile_id)
```

where:

- `shared_shaper_id`: Shared shaper ID to be created.
- `shaper_profile id`: Shaper profile ID for shared shaper.

Set port traffic management shared shaper

Update the port traffic management shared shaper:

```
testpmd> set port tm node shared shaper (port_id) (shared_shaper_id) \
(shaper_profile_id)
```

where:

- `shared_shaper_id`: Shared shaper ID to be update.
- `shaper_profile id`: Shaper profile ID for shared shaper.

Delete port traffic management shared shaper

Delete the port traffic management shared shaper:

```
testpmd> del port tm node shared shaper (port_id) (shared_shaper_id)
```

where:

- `shared_shaper_id`: Shared shaper ID to be deleted.

Set port traffic management hierarchy node private shaper

set the port traffic management hierarchy node private shaper:

```
testpmd> set port tm node shaper profile (port_id) (node_id) \
(shaper_profile_id)
```

where:

- `shaper_profile id`: Private shaper profile ID to be enabled on the hierarchy node.

Add port traffic management WRED profile

Create a new WRED profile:

```
testpmd> add port tm node wred profile (port_id) (wred_profile_id) \
(color_g) (min_th_g) (max_th_g) (maxp_inv_g) (wq_log2_g) \
(color_y) (min_th_y) (max_th_y) (maxp_inv_y) (wq_log2_y) \
(color_r) (min_th_r) (max_th_r) (maxp_inv_r) (wq_log2_r)
```

where:

- `wred_profile id`: Identifier for the newly create WRED profile
- `color_g`: Packet color (green)
- `min_th_g`: Minimum queue threshold for packet with green color
- `max_th_g`: Minimum queue threshold for packet with green color
- `maxp_inv_g`: Inverse of packet marking probability maximum value (maxp)
- `wq_log2_g`: Negated log2 of queue weight (wq)
- `color_y`: Packet color (yellow)
- `min_th_y`: Minimum queue threshold for packet with yellow color
- `max_th_y`: Minimum queue threshold for packet with yellow color
- `maxp_inv_y`: Inverse of packet marking probability maximum value (maxp)
- `wq_log2_y`: Negated log2 of queue weight (wq)
- `color_r`: Packet color (red)
- `min_th_r`: Minimum queue threshold for packet with yellow color
- `max_th_r`: Minimum queue threshold for packet with yellow color
- `maxp_inv_r`: Inverse of packet marking probability maximum value (maxp)
- `wq_log2_r`: Negated log2 of queue weight (wq)

Delete port traffic management WRED profile

Delete the WRED profile:

```
testpmd> del port tm node wred profile (port_id) (wred_profile_id)
```

Add port traffic management hierarchy nonleaf node

Add nonleaf node to port traffic management hierarchy:

```
testpmd> add port tm nonleaf node (port_id) (node_id) (parent_node_id) \
(priority) (weight) (level_id) (shaper_profile_id) \
(n_sp_priorities) (stats_mask) (n_shared_shapers) \
[(shared_shaper_0) (shared_shaper_1) ...] \
```

where:

- `parent_node_id`: Node ID of the parent.
- `priority`: Node priority (highest node priority is zero). This is used by the SP algorithm running on the parent node for scheduling this node.
- `weight`: Node weight (lowest weight is one). The node weight is relative to the weight sum of all siblings that have the same priority. It is used by the WFQ algorithm running on the parent node for scheduling this node.
- `level_id`: Hierarchy level of the node.
- `shaper_profile_id`: Shaper profile ID of the private shaper to be used by the node.
- `n_sp_priorities`: Number of strict priorities.
- `stats_mask`: Mask of statistics counter types to be enabled for this node.
- `n_shared_shapers`: Number of shared shapers.
- `shared_shaper_id`: Shared shaper id.

Add port traffic management hierarchy leaf node

Add leaf node to port traffic management hierarchy:

```
testpmd> add port tm leaf node (port_id) (node_id) (parent_node_id) \
(priority) (weight) (level_id) (shaper_profile_id) \
(cman_mode) (wred_profile_id) (stats_mask) (n_shared_shapers) \
[(shared_shaper_id) (shared_shaper_id) ...] \
```

where:

- `parent_node_id`: Node ID of the parent.
- `priority`: Node priority (highest node priority is zero). This is used by the SP algorithm running on the parent node for scheduling this node.
- `weight`: Node weight (lowest weight is one). The node weight is relative to the weight sum of all siblings that have the same priority. It is used by the WFQ algorithm running on the parent node for scheduling this node.

- `level_id`: Hierarchy level of the node.
- `shaper_profile_id`: Shaper profile ID of the private shaper to be used by the node.
- `cman_mode`: Congestion management mode to be enabled for this node.
- `wred_profile_id`: WRED profile id to be enabled for this node.
- `stats_mask`: Mask of statistics counter types to be enabled for this node.
- `n_shared_shapers`: Number of shared shapers.
- `shared_shaper_id`: Shared shaper id.

Delete port traffic management hierarchy node

Delete node from port traffic management hierarchy:

```
testpmd> del port tm node (port_id) (node_id)
```

Update port traffic management hierarchy parent node

Update port traffic management hierarchy parent node:

```
testpmd> set port tm node parent (port_id) (node_id) (parent_node_id) \
(priority) (weight)
```

This function can only be called after the hierarchy commit invocation. Its success depends on the port support for this operation, as advertised through the port capability set. This function is valid for all nodes of the traffic management hierarchy except root node.

Suspend port traffic management hierarchy node

testpmd> suspend port tm node (port_id) (node_id)

Resume port traffic management hierarchy node

testpmd> resume port tm node (port_id) (node_id)

Commit port traffic management hierarchy

Commit the traffic management hierarchy on the port:

```
testpmd> port tm hierarchy commit (port_id) (clean_on_fail)
```

where:

- `clean_on_fail`: When set to non-zero, hierarchy is cleared on function call failure. On the other hand, hierarchy is preserved when this parameter is equal to zero.

Set port traffic management mark VLAN dei

Enables/Disables the traffic management marking on the port for VLAN packets:

```
testpmd> set port tm mark vlan_dei <port_id> <green> <yellow> <red>
```

where:

- `port_id`: The port which on which VLAN packets marked as `green` or `yellow` or `red` will have dei bit enabled
- `green` enable 1, disable 0 marking for dei bit of VLAN packets marked as green
- `yellow` enable 1, disable 0 marking for dei bit of VLAN packets marked as yellow
- `red` enable 1, disable 0 marking for dei bit of VLAN packets marked as red

Set port traffic management mark IP dscp

Enables/Disables the traffic management marking on the port for IP dscp packets:

```
testpmd> set port tm mark ip_dscp <port_id> <green> <yellow> <red>
```

where:

- `port_id`: The port which on which IP packets marked as `green` or `yellow` or `red` will have IP dscp bits updated
- `green` enable 1, disable 0 marking IP dscp to low drop precedence for green packets
- `yellow` enable 1, disable 0 marking IP dscp to medium drop precedence for yellow packets
- `red` enable 1, disable 0 marking IP dscp to high drop precedence for red packets

Set port traffic management mark IP ecn

Enables/Disables the traffic management marking on the port for IP ecn packets:

```
testpmd> set port tm mark ip_ecn <port_id> <green> <yellow> <red>
```

where:

- `port_id`: The port which on which IP packets marked as `green` or `yellow` or `red` will have IP ecn bits updated
- `green` enable 1, disable 0 marking IP ecn for green marked packets with ecn of 2'b01 or 2'b10 to ecn of 2'b11 when IP is caring TCP or SCTP
- `yellow` enable 1, disable 0 marking IP ecn for yellow marked packets with ecn of 2'b01 or 2'b10 to ecn of 2'b11 when IP is caring TCP or SCTP
- `red` enable 1, disable 0 marking IP ecn for yellow marked packets with ecn of 2'b01 or 2'b10 to ecn of 2'b11 when IP is caring TCP or SCTP

Set port traffic management default hierarchy (softnic forwarding mode)

set the traffic management default hierarchy on the port:

```
testpmd> set port tm hierarchy default (port_id)
```

8.4.12 Filter Functions

This section details the available filter functions that are available.

Note these functions interface the deprecated legacy filtering framework, superseded by *rte_flow*. See *Flow rules management*.

ethertype_filter

Add or delete a L2 Ethertype filter, which identify packets by their L2 Ethertype mainly assign them to a receive queue:

```
ethertype_filter (port_id) (add|del) (mac_addr|mac_ignr) (mac_address) \
                 ethertype (ether_type) (drop|fwd) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the Ethertype filter assigned on.
- `mac_addr`: Compare destination mac address.
- `mac_ignr`: Ignore destination mac address match.
- `mac_address`: Destination mac address to match.
- `ether_type`: The EtherType value want to match, for example 0x0806 for ARP packet. 0x0800 (IPv4) and 0x86DD (IPv6) are invalid.
- `queue_id`: The receive queue associated with this EtherType filter. It is meaningless when deleting or dropping.

Example, to add/remove an ethertype filter rule:

```
testpmd> ethertype_filter 0 add mac_ignr 00:11:22:33:44:55 \
                           ethertype 0x0806 fwd queue 3

testpmd> ethertype_filter 0 del mac_ignr 00:11:22:33:44:55 \
                           ethertype 0x0806 fwd queue 3
```

2tuple_filter

Add or delete a 2-tuple filter, which identifies packets by specific protocol and destination TCP/UDP port and forwards packets into one of the receive queues:

```
2tuple_filter (port_id) (add|del) dst_port (dst_port_value) \
              protocol (protocol_value) mask (mask_value) \
              tcp_flags (tcp_flags_value) priority (prio_value) \
              queue (queue_id)
```

The available information parameters are:

- port_id: The port which the 2-tuple filter assigned on.
- dst_port_value: Destination port in L4.
- protocol_value: IP L4 protocol.
- mask_value: Participates in the match or not by bit for field above, 1b means participate.
- tcp_flags_value: TCP control bits. The non-zero value is invalid, when the pro_value is not set to 0x06 (TCP).
- prio_value: Priority of this filter.
- queue_id: The receive queue associated with this 2-tuple filter.

Example, to add/remove an 2tuple filter rule:

```
testpmd> 2tuple_filter 0 add dst_port 32 protocol 0x06 mask 0x03 \
                       tcp_flags 0x02 priority 3 queue 3

testpmd> 2tuple_filter 0 del dst_port 32 protocol 0x06 mask 0x03 \
                       tcp_flags 0x02 priority 3 queue 3
```

5tuple_filter

Add or delete a 5-tuple filter, which consists of a 5-tuple (protocol, source and destination IP addresses, source and destination TCP/UDP/SCTP port) and routes packets into one of the receive queues:

```
5tuple_filter (port_id) (add|del) dst_ip (dst_address) src_ip \
              (src_address) dst_port (dst_port_value) \
              src_port (src_port_value) protocol (protocol_value) \
              mask (mask_value) tcp_flags (tcp_flags_value) \
              priority (prio_value) queue (queue_id)
```

The available information parameters are:

- port_id: The port which the 5-tuple filter assigned on.
- dst_address: Destination IP address.
- src_address: Source IP address.
- dst_port_value: TCP/UDP destination port.
- src_port_value: TCP/UDP source port.
- protocol_value: L4 protocol.
- mask_value: Participates in the match or not by bit for field above, 1b means participate
- tcp_flags_value: TCP control bits. The non-zero value is invalid, when the protocol_value is not set to 0x06 (TCP).
- prio_value: The priority of this filter.
- queue_id: The receive queue associated with this 5-tuple filter.

Example, to add/remove an 5tuple filter rule:

```
testpmd> 5tuple_filter 0 add dst_ip 2.2.2.5 src_ip 2.2.2.4 \
         dst_port 64 src_port 32 protocol 0x06 mask 0x1F \
         flags 0x0 priority 3 queue 3

testpmd> 5tuple_filter 0 del dst_ip 2.2.2.5 src_ip 2.2.2.4 \
         dst_port 64 src_port 32 protocol 0x06 mask 0x1F \
         flags 0x0 priority 3 queue 3
```

syn_filter

Using the SYN filter, TCP packets whose *SYN* flag is set can be forwarded to a separate queue:

```
syn_filter (port_id) (add|del) priority (high|low) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the SYN filter assigned on.
- `high`: This SYN filter has higher priority than other filters.
- `low`: This SYN filter has lower priority than other filters.
- `queue_id`: The receive queue associated with this SYN filter

Example:

```
testpmd> syn_filter 0 add priority high queue 3
```

flex_filter

With flex filter, packets can be recognized by any arbitrary pattern within the first 128 bytes of the packet and routed into one of the receive queues:

```
flex_filter (port_id) (add|del) len (len_value) bytes (bytes_value) \
            mask (mask_value) priority (prio_value) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the Flex filter is assigned on.
- `len_value`: Filter length in bytes, no greater than 128.
- `bytes_value`: A string in hexadecimal, means the value the flex filter needs to match.
- `mask_value`: A string in hexadecimal, bit 1 means corresponding byte participates in the match.
- `prio_value`: The priority of this filter.
- `queue_id`: The receive queue associated with this Flex filter.

Example:

```
testpmd> flex_filter 0 add len 16 bytes 0x00000000000000000000000008060000 \
                       mask 000C priority 3 queue 3

testpmd> flex_filter 0 del len 16 bytes 0x00000000000000000000000008060000 \
                       mask 000C priority 3 queue 3
```

flow_director_filter

The Flow Director works in receive mode to identify specific flows or sets of flows and route them to specific queues.

Four types of filtering are supported which are referred to as Perfect Match, Signature, Perfect-mac-vlan and Perfect-tunnel filters, the match mode is set by the `--pkt-filter-mode` command-line parameter:

- Perfect match filters. The hardware checks a match between the masked fields of the received packets and the programmed filters. The masked fields are for IP flow.
- Signature filters. The hardware checks a match between a hash-based signature of the masked fields of the received packet.
- Perfect-mac-vlan match filters. The hardware checks a match between the masked fields of the received packets and the programmed filters. The masked fields are for MAC VLAN flow.
- Perfect-tunnel match filters. The hardware checks a match between the masked fields of the received packets and the programmed filters. The masked fields are for tunnel flow.
- Perfect-raw-flow-type match filters. The hardware checks a match between the masked fields of the received packets and pre-loaded raw (template) packet. The masked fields are specified by input sets.

The Flow Director filters can match the different fields for different type of packet: flow type, specific input set per flow type and the flexible payload.

The Flow Director can also mask out parts of all of these fields so that filters are only applied to certain fields or parts of the fields.

Note that for raw flow type mode the source and destination fields in the raw packet buffer need to be presented in a reversed order with respect to the expected received packets. For example: IP source and destination addresses or TCP/UDP/SCTP source and destination ports

Different NICs may have different capabilities, command show port fdir (port_id) can be used to acquire the information.

Commands to add flow director filters of different flow types:

```
flow_director_filter (port_id) mode IP (add|del|update) \
                     flow (ipv4-other|ipv4-frag|ipv6-other|ipv6-frag) \
                     src (src_ip_address) dst (dst_ip_address) \
                     tos (tos_value) proto (proto_value) ttl (ttl_value) \
                     vlan (vlan_value) flexbytes (flexbytes_value) \
                     (drop|fwd) pf|vf(vf_id) queue (queue_id) \
                     fd_id (fd_id_value)

flow_director_filter (port_id) mode IP (add|del|update) \
                     flow (ipv4-tcp|ipv4-udp|ipv6-tcp|ipv6-udp) \
                     src (src_ip_address) (src_port) \
                     dst (dst_ip_address) (dst_port) \
                     tos (tos_value) ttl (ttl_value) \
                     vlan (vlan_value) flexbytes (flexbytes_value) \
                     (drop|fwd) queue pf|vf(vf_id) (queue_id) \
                     fd_id (fd_id_value)

flow_director_filter (port_id) mode IP (add|del|update) \
                     flow (ipv4-sctp|ipv6-sctp) \
                     src (src_ip_address) (src_port) \
                     dst (dst_ip_address) (dst_port) \
```

(continues on next page)

(continued from previous page)

```
                    tos (tos_value) ttl (ttl_value) \
                    tag (verification_tag) vlan (vlan_value) \
                    flexbytes (flexbytes_value) (drop|fwd) \
                    pf|vf(vf_id) queue (queue_id) fd_id (fd_id_value)

flow_director_filter (port_id) mode IP (add|del|update) flow l2_payload \
                    ether (ethertype) flexbytes (flexbytes_value) \
                    (drop|fwd) pf|vf(vf_id) queue (queue_id)
                    fd_id (fd_id_value)

flow_director_filter (port_id) mode MAC-VLAN (add|del|update) \
                    mac (mac_address) vlan (vlan_value) \
                    flexbytes (flexbytes_value) (drop|fwd) \
                    queue (queue_id) fd_id (fd_id_value)

flow_director_filter (port_id) mode Tunnel (add|del|update) \
                    mac (mac_address) vlan (vlan_value) \
                    tunnel (NVGRE|VxLAN) tunnel-id (tunnel_id_value) \
                    flexbytes (flexbytes_value) (drop|fwd) \
                    queue (queue_id) fd_id (fd_id_value)

flow_director_filter (port_id) mode raw (add|del|update) flow (flow_id) \
                    (drop|fwd) queue (queue_id) fd_id (fd_id_value) \
                    packet (packet file name)
```

For example, to add an ipv4-udp flow type filter:

```
testpmd> flow_director_filter 0 mode IP add flow ipv4-udp src 2.2.2.3 32 \
         dst 2.2.2.5 33 tos 2 ttl 40 vlan 0x1 flexbytes (0x88,0x48) \
         fwd pf queue 1 fd_id 1
```

For example, add an ipv4-other flow type filter:

```
testpmd> flow_director_filter 0 mode IP add flow ipv4-other src 2.2.2.3 \
          dst 2.2.2.5 tos 2 proto 20 ttl 40 vlan 0x1 \
          flexbytes (0x88,0x48) fwd pf queue 1 fd_id 1
```

flush_flow_director

Flush all flow director filters on a device:

```
testpmd> flush_flow_director (port_id)
```

Example, to flush all flow director filter on port 0:

```
testpmd> flush_flow_director 0
```

flow_director_mask

Set flow director's input masks:

```
flow_director_mask (port_id) mode IP vlan (vlan_value) \
                   src_mask (ipv4_src) (ipv6_src) (src_port) \
                   dst_mask (ipv4_dst) (ipv6_dst) (dst_port)

flow_director_mask (port_id) mode MAC-VLAN vlan (vlan_value)

flow_director_mask (port_id) mode Tunnel vlan (vlan_value) \
                   mac (mac_value) tunnel-type (tunnel_type_value) \
                   tunnel-id (tunnel_id_value)
```

Example, to set flow director mask on port 0:

```
testpmd> flow_director_mask 0 mode IP vlan 0xefff \
         src_mask 255.255.255.255 \
             FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF 0xFFFF \
         dst_mask 255.255.255.255 \
             FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF 0xFFFF
```

flow_director_flex_mask

set masks of flow director's flexible payload based on certain flow type:

```
testpmd> flow_director_flex_mask (port_id) \
         flow (none|ipv4-other|ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp| \
               ipv6-other|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp| \
               l2_payload|all) (mask)
```

Example, to set flow director's flex mask for all flow type on port 0:

```
testpmd> flow_director_flex_mask 0 flow all \
         (0xff,0xff,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

flow_director_flex_payload

Configure flexible payload selection:

```
flow_director_flex_payload (port_id) (raw|l2|l3|l4) (config)
```

For example, to select the first 16 bytes from the offset 4 (bytes) of packet's payload as flexible payload:

```
testpmd> flow_director_flex_payload 0 l4 \
         (4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19)
```

get_sym_hash_ena_per_port

Get symmetric hash enable configuration per port:

```
get_sym_hash_ena_per_port (port_id)
```

For example, to get symmetric hash enable configuration of port 1:

```
testpmd> get_sym_hash_ena_per_port 1
```

set_sym_hash_ena_per_port

Set symmetric hash enable configuration per port to enable or disable:

```
set_sym_hash_ena_per_port (port_id) (enable|disable)
```

For example, to set symmetric hash enable configuration of port 1 to enable:

```
testpmd> set_sym_hash_ena_per_port 1 enable
```

get_hash_global_config

Get the global configurations of hash filters:

```
get_hash_global_config (port_id)
```

For example, to get the global configurations of hash filters of port 1:

```
testpmd> get_hash_global_config 1
```

set_hash_global_config

Set the global configurations of hash filters:

```
set_hash_global_config (port_id) (toeplitz|simple_xor|symmetric_toeplitz|default) \
(ipv4|ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp|ipv4-other|ipv6|ipv6-frag| \
ipv6-tcp|ipv6-udp|ipv6-sctp|ipv6-other|l2_payload|<flow_id>) \
(enable|disable)
```

For example, to enable simple_xor for flow type of ipv6 on port 2:

```
testpmd> set_hash_global_config 2 simple_xor ipv6 enable
```

set_hash_input_set

Set the input set for hash:

```
set_hash_input_set (port_id) (ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp| \
ipv4-other|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp|ipv6-other| \
l2_payload|<flow_id>) (ovlan|ivlan|src-ipv4|dst-ipv4|src-ipv6|dst-ipv6| \
ipv4-tos|ipv4-proto|ipv6-tc|ipv6-next-header|udp-src-port|udp-dst-port| \
tcp-src-port|tcp-dst-port|sctp-src-port|sctp-dst-port|sctp-veri-tag| \
udp-key|gre-key|fld-1st|fld-2nd|fld-3rd|fld-4th|fld-5th|fld-6th|fld-7th| \
fld-8th|none) (select|add)
```

For example, to add source IP to hash input set for flow type of ipv4-udp on port 0:

```
testpmd> set_hash_input_set 0 ipv4-udp src-ipv4 add
```

set_fdir_input_set

The Flow Director filters can match the different fields for different type of packet, i.e. specific input set on per flow type and the flexible payload. This command can be used to change input set for each flow type.

Set the input set for flow director:

```
set_fdir_input_set (port_id) (ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp| \
ipv4-other|ipv6|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp|ipv6-other| \
l2_payload|<flow_id>) (ivlan|ethertype|src-ipv4|dst-ipv4|src-ipv6|dst-ipv6| \
ipv4-tos|ipv4-proto|ipv4-ttl|ipv6-tc|ipv6-next-header|ipv6-hop-limits| \
tudp-src-port|udp-dst-port|cp-src-port|tcp-dst-port|sctp-src-port| \
sctp-dst-port|sctp-veri-tag|none) (select|add)
```

For example to add source IP to FD input set for flow type of ipv4-udp on port 0:

```
testpmd> set_fdir_input_set 0 ipv4-udp src-ipv4 add
```

global_config

Set different GRE key length for input set:

```
global_config (port_id) gre-key-len (number in bytes)
```

For example to set GRE key length for input set to 4 bytes on port 0:

```
testpmd> global_config 0 gre-key-len 4
```

8.4.13 Flow rules management

Control of the generic flow API (*rte_flow*) is fully exposed through the `flow` command (validation, creation, destruction, queries and operation modes).

Considering *rte_flow* overlaps with all *Filter Functions*, using both features simultaneously may cause undefined side-effects and is therefore not recommended.

`flow` syntax

Because the `flow` command uses dynamic tokens to handle the large number of possible flow rules combinations, its behavior differs slightly from other commands, in particular:

- Pressing *?* or the *<tab>* key displays contextual help for the current token, not that of the entire command.
- Optional and repeated parameters are supported (provided they are listed in the contextual help).

The first parameter stands for the operation mode. Possible operations and their general syntax are described below. They are covered in detail in the following sections.

- Check whether a flow rule can be created:

```
flow validate {port_id}
    [group {group_id}] [priority {level}] [ingress] [egress] [transfer]
    pattern {item} [/ {item} [...]] / end
    actions {action} [/ {action} [...]] / end
```

- Create a flow rule:

```
flow create {port_id}
    [group {group_id}] [priority {level}] [ingress] [egress] [transfer]
    pattern {item} [/ {item} [...]] / end
    actions {action} [/ {action} [...]] / end
```

- Destroy specific flow rules:

```
flow destroy {port_id} rule {rule_id} [...]
```

- Destroy all flow rules:

```
flow flush {port_id}
```

- Query an existing flow rule:

```
flow query {port_id} {rule_id} {action}
```

- List existing flow rules sorted by priority, filtered by group identifiers:

```
flow list {port_id} [group {group_id}] [...]
```

- Restrict ingress traffic to the defined flow rules:

```
flow isolate {port_id} {boolean}
```

- Dump internal representation information of all flows in hardware:

```
flow dump {port_id} {output_file}
```

- List and destroy aged flow rules:

```
flow aged {port_id} [destroy]
```

Validating flow rules

`flow validate` reports whether a flow rule would be accepted by the underlying device in its current state but stops short of creating it. It is bound to `rte_flow_validate()`:

```
flow validate {port_id}
   [group {group_id}] [priority {level}] [ingress] [egress] [transfer]
   pattern {item} [/ {item} [...]] / end
   actions {action} [/ {action} [...]] / end
```

If successful, it will show:

```
Flow rule validated
```

Otherwise it will show an error message of the form:

```
Caught error type [...] ([...]): [...]
```

This command uses the same parameters as `flow create`, their format is described in *Creating flow rules*.

Check whether redirecting any Ethernet packet received on port 0 to RX queue index 6 is supported:

```
testpmd> flow validate 0 ingress pattern eth / end
   actions queue index 6 / end
Flow rule validated
testpmd>
```

Port 0 does not support TCPv6 rules:

```
testpmd> flow validate 0 ingress pattern eth / ipv6 / tcp / end
   actions drop / end
Caught error type 9 (specific pattern item): Invalid argument
testpmd>
```

Creating flow rules

`flow create` validates and creates the specified flow rule. It is bound to `rte_flow_create()`:

```
flow create {port_id}
   [group {group_id}] [priority {level}] [ingress] [egress] [transfer]
   pattern {item} [/ {item} [...]] / end
   actions {action} [/ {action} [...]] / end
```

If successful, it will return a flow rule ID usable with other commands:

```
Flow rule #[...] created
```

Otherwise it will show an error message of the form:

```
Caught error type [...] ([...]): [...]
```

Parameters describe in the following order:

- Attributes (*group*, *priority*, *ingress*, *egress*, *transfer* tokens).
- A matching pattern, starting with the *pattern* token and terminated by an *end* pattern item.
- Actions, starting with the *actions* token and terminated by an *end* action.

These translate directly to *rte_flow* objects provided as-is to the underlying functions.

The shortest valid definition only comprises mandatory tokens:

```
testpmd> flow create 0 pattern end actions end
```

Note that PMDs may refuse rules that essentially do nothing such as this one.

All unspecified object values are automatically initialized to 0.

Attributes

These tokens affect flow rule attributes (`struct rte_flow_attr`) and are specified before the `pattern` token.

- `group {group id}`: priority group.
- `priority {level}`: priority level within group.
- `ingress`: rule applies to ingress traffic.
- `egress`: rule applies to egress traffic.
- `transfer`: apply rule directly to endpoints found in pattern.

Each instance of an attribute specified several times overrides the previous value as shown below (group 4 is used):

```
testpmd> flow create 0 group 42 group 24 group 4 [...]
```

Note that once enabled, `ingress` and `egress` cannot be disabled.

While not specifying a direction is an error, some rules may allow both simultaneously.

Most rules affect RX therefore contain the `ingress` token:

```
testpmd> flow create 0 ingress pattern [...]
```

Matching pattern

A matching pattern starts after the `pattern` token. It is made of pattern items and is terminated by a mandatory `end` item.

Items are named after their type (*RTE_FLOW_ITEM_TYPE_* from `enum rte_flow_item_type`).

The `/` token is used as a separator between pattern items as shown below:

```
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / end [...]
```

Note that protocol items like these must be stacked from lowest to highest layer to make sense. For instance, the following rule is either invalid or unlikely to match any packet:

```
testpmd> flow create 0 ingress pattern eth / udp / ipv4 / end [...]
```

More information on these restrictions can be found in the *rte_flow* documentation.

Several items support additional specification structures, for example `ipv4` allows specifying source and destination addresses as follows:

```
testpmd> flow create 0 ingress pattern eth / ipv4 src is 10.1.1.1
   dst is 10.2.0.0 / end [...]
```

This rule matches all IPv4 traffic with the specified properties.

In this example, `src` and `dst` are field names of the underlying `struct rte_flow_item_ipv4` object. All item properties can be specified in a similar fashion.

The `is` token means that the subsequent value must be matched exactly, and assigns `spec` and `mask` fields in `struct rte_flow_item` accordingly. Possible assignment tokens are:

- `is`: match value perfectly (with full bit-mask).
- `spec`: match value according to configured bit-mask.
- `last`: specify upper bound to establish a range.
- `mask`: specify bit-mask with relevant bits set to one.
- `prefix`: generate bit-mask with <prefix-length> most-significant bits set to one.

These yield identical results:

```
ipv4 src is 10.1.1.1
```

```
ipv4 src spec 10.1.1.1 src mask 255.255.255.255
```

```
ipv4 src spec 10.1.1.1 src prefix 32
```

```
ipv4 src is 10.1.1.1 src last 10.1.1.1 # range with a single value
```

```
ipv4 src is 10.1.1.1 src last 0 # 0 disables range
```

Inclusive ranges can be defined with `last`:

```
ipv4 src is 10.1.1.1 src last 10.2.3.4 # 10.1.1.1 to 10.2.3.4
```

Note that `mask` affects both `spec` and `last`:

```
ipv4 src is 10.1.1.1 src last 10.2.3.4 src mask 255.255.0.0
   # matches 10.1.0.0 to 10.2.255.255
```

Properties can be modified multiple times:

```
ipv4 src is 10.1.1.1 src is 10.1.2.3 src is 10.2.3.4 # matches 10.2.3.4
```

```
ipv4 src is 10.1.1.1 src prefix 24 src prefix 16 # matches 10.1.0.0/16
```

Pattern items

This section lists supported pattern items and their attributes, if any.

- `end`: end list of pattern items.
- `void`: no-op pattern item.
- `invert`: perform actions when pattern does not match.
- `any`: match any protocol for the current layer.
  - `num {unsigned}`: number of layers covered.
- `pf`: match traffic from/to the physical function.
- `vf`: match traffic from/to a virtual function ID.
  - `id {unsigned}`: VF ID.
- `phy_port`: match traffic from/to a specific physical port.
  - `index {unsigned}`: physical port index.
- `port_id`: match traffic from/to a given DPDK port ID.
  - `id {unsigned}`: DPDK port ID.
- `mark`: match value set in previously matched flow rule using the mark action.
  - `id {unsigned}`: arbitrary integer value.
- `raw`: match an arbitrary byte string.
  - `relative {boolean}`: look for pattern after the previous item.
  - `search {boolean}`: search pattern from offset (see also limit).
  - `offset {integer}`: absolute or relative offset for pattern.
  - `limit {unsigned}`: search area limit for start of pattern.
  - `pattern {string}`: byte string to look for.
- `eth`: match Ethernet header.
  - `dst {MAC-48}`: destination MAC.
  - `src {MAC-48}`: source MAC.
  - `type {unsigned}`: EtherType or TPID.
- `vlan`: match 802.1Q/ad VLAN tag.
  - `tci {unsigned}`: tag control information.
  - `pcp {unsigned}`: priority code point.
  - `dei {unsigned}`: drop eligible indicator.
  - `vid {unsigned}`: VLAN identifier.

  - `inner_type {unsigned}`: inner EtherType or TPID.

- `ipv4`: match IPv4 header.
  - `tos {unsigned}`: type of service.
  - `ttl {unsigned}`: time to live.
  - `proto {unsigned}`: next protocol ID.
  - `src {ipv4 address}`: source address.
  - `dst {ipv4 address}`: destination address.

- `ipv6`: match IPv6 header.
  - `tc {unsigned}`: traffic class.
  - `flow {unsigned}`: flow label.
  - `proto {unsigned}`: protocol (next header).
  - `hop {unsigned}`: hop limit.
  - `src {ipv6 address}`: source address.
  - `dst {ipv6 address}`: destination address.

- `icmp`: match ICMP header.
  - `type {unsigned}`: ICMP packet type.
  - `code {unsigned}`: ICMP packet code.

- `udp`: match UDP header.
  - `src {unsigned}`: UDP source port.
  - `dst {unsigned}`: UDP destination port.

- `tcp`: match TCP header.
  - `src {unsigned}`: TCP source port.
  - `dst {unsigned}`: TCP destination port.

- `sctp`: match SCTP header.
  - `src {unsigned}`: SCTP source port.
  - `dst {unsigned}`: SCTP destination port.
  - `tag {unsigned}`: validation tag.
  - `cksum {unsigned}`: checksum.

- `vxlan`: match VXLAN header.
  - `vni {unsigned}`: VXLAN identifier.

- `e_tag`: match IEEE 802.1BR E-Tag header.
  - `grp_ecid_b {unsigned}`: GRP and E-CID base.

- `nvgre`: match NVGRE header.
  - `tni {unsigned}`: virtual subnet ID.

- `mpls`: match MPLS header.
  - `label {unsigned}`: MPLS label.
- `gre`: match GRE header.
  - `protocol {unsigned}`: protocol type.
- `gre_key`: match GRE optional key field.
  - `value {unsigned}`: key value.
- `fuzzy`: fuzzy pattern match, expect faster than default.
  - `thresh {unsigned}`: accuracy threshold.
- `gtp`, `gtpc`, `gtpu`: match GTPv1 header.
  - `teid {unsigned}`: tunnel endpoint identifier.
- `geneve`: match GENEVE header.
  - `vni {unsigned}`: virtual network identifier.
  - `protocol {unsigned}`: protocol type.
- `vxlan-gpe`: match VXLAN-GPE header.
  - `vni {unsigned}`: VXLAN-GPE identifier.
- `arp_eth_ipv4`: match ARP header for Ethernet/IPv4.
  - `sha {MAC-48}`: sender hardware address.
  - `spa {ipv4 address}`: sender IPv4 address.
  - `tha {MAC-48}`: target hardware address.
  - `tpa {ipv4 address}`: target IPv4 address.
- `ipv6_ext`: match presence of any IPv6 extension header.
  - `next_hdr {unsigned}`: next header.
- `icmp6`: match any ICMPv6 header.
  - `type {unsigned}`: ICMPv6 type.
  - `code {unsigned}`: ICMPv6 code.
- `icmp6_nd_ns`: match ICMPv6 neighbor discovery solicitation.
  - `target_addr {ipv6 address}`: target address.
- `icmp6_nd_na`: match ICMPv6 neighbor discovery advertisement.
  - `target_addr {ipv6 address}`: target address.
- `icmp6_nd_opt`: match presence of any ICMPv6 neighbor discovery option.
  - `type {unsigned}`: ND option type.
- `icmp6_nd_opt_sla_eth`: match ICMPv6 neighbor discovery source Ethernet link-layer address option.
  - `sla {MAC-48}`: source Ethernet LLA.

- `icmp6_nd_opt_tla_eth`: match ICMPv6 neighbor discovery target Ethernet link-layer address option.
  - `tla {MAC-48}`: target Ethernet LLA.
- `meta`: match application specific metadata.
  - `data {unsigned}`: metadata value.
- `gtp_psc`: match GTP PDU extension header with type 0x85.
  - `pdu_type {unsigned}`: PDU type.
  - `qfi {unsigned}`: QoS flow identifier.
- `pppoes`, `pppoed`: match PPPoE header.
  - `session_id {unsigned}`: session identifier.
- `pppoe_proto_id`: match PPPoE session protocol identifier.
  - `proto_id {unsigned}`: PPP protocol identifier.
- `l2tpv3oip`: match L2TPv3 over IP header.
  - `session_id {unsigned}`: L2TPv3 over IP session identifier.
- `ah`: match AH header.
  - `spi {unsigned}`: security parameters index.
- `pfcp`: match PFCP header.
  - `s_field {unsigned}`: S field.
  - `seid {unsigned}`: session endpoint identifier.

Actions list

A list of actions starts after the `actions` token in the same fashion as *Matching pattern*; actions are separated by `/` tokens and the list is terminated by a mandatory `end` action.

Actions are named after their type (*RTE_FLOW_ACTION_TYPE_* from `enum rte_flow_action_type`).

Dropping all incoming UDPv4 packets can be expressed as follows:

```
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / end
   actions drop / end
```

Several actions have configurable properties which must be specified when there is no valid default value. For example, `queue` requires a target queue index.

This rule redirects incoming UDPv4 traffic to queue index 6:

```
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / end
   actions queue index 6 / end
```

While this one could be rejected by PMDs (unspecified queue index):

```
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / end
   actions queue / end
```

As defined by *rte_flow*, the list is not ordered, all actions of a given rule are performed simultaneously. These are equivalent:

```
queue index 6 / void / mark id 42 / end
```

```
void / mark id 42 / queue index 6 / end
```

All actions in a list should have different types, otherwise only the last action of a given type is taken into account:

```
queue index 4 / queue index 5 / queue index 6 / end # will use queue 6
```

```
drop / drop / drop / end # drop is performed only once
```

```
mark id 42 / queue index 3 / mark id 24 / end # mark will be 24
```

Considering they are performed simultaneously, opposite and overlapping actions can sometimes be combined when the end result is unambiguous:

```
drop / queue index 6 / end # drop has no effect
```

```
queue index 6 / rss queues 6 7 8 / end # queue has no effect
```

```
drop / passthru / end # drop has no effect
```

Note that PMDs may still refuse such combinations.

Actions

This section lists supported actions and their attributes, if any.

- `end`: end list of actions.
- `void`: no-op action.
- `passthru`: let subsequent rule process matched packets.
- `jump`: redirect traffic to group on device.
  - `group {unsigned}`: group to redirect to.
- `mark`: attach 32 bit value to packets.
  - `id {unsigned}`: 32 bit value to return with packets.
- `flag`: flag packets.
- `queue`: assign packets to a given queue index.
  - `index {unsigned}`: queue index to use.
- `drop`: drop packets (note: passthru has priority).
- `count`: enable counters for this rule.
- `rss`: spread packets among several queues.

  - `func {hash function}`: RSS hash function to apply, allowed tokens are the same as *set_hash_global_config*.
  - `level {unsigned}`: encapsulation level for `types`.
  - `types [{RSS hash type} [...]] end`: specific RSS hash types, allowed tokens are the same as *set_hash_input_set*, except that an empty list does not disable RSS but instead requests unspecified "best-effort" settings.
  - `key {string}`: RSS hash key, overrides `key_len`.
  - `key_len {unsigned}`: RSS hash key length in bytes, can be used in conjunction with `key` to pad or truncate it.
  - `queues [{unsigned} [...]] end`: queue indices to use.
- `pf`: direct traffic to physical function.
- `vf`: direct traffic to a virtual function ID.
  - `original {boolean}`: use original VF ID if possible.
  - `id {unsigned}`: VF ID.
- `phy_port`: direct packets to physical port index.
  - `original {boolean}`: use original port index if possible.
  - `index {unsigned}`: physical port index.
- `port_id`: direct matching traffic to a given DPDK port ID.
  - `original {boolean}`: use original DPDK port ID if possible.
  - `id {unsigned}`: DPDK port ID.
- `of_set_mpls_ttl`: OpenFlow's `OFPAT_SET_MPLS_TTL`.
  - `mpls_ttl`: MPLS TTL.
- `of_dec_mpls_ttl`: OpenFlow's `OFPAT_DEC_MPLS_TTL`.
- `of_set_nw_ttl`: OpenFlow's `OFPAT_SET_NW_TTL`.
  - `nw_ttl`: IP TTL.
- `of_dec_nw_ttl`: OpenFlow's `OFPAT_DEC_NW_TTL`.
- `of_copy_ttl_out`: OpenFlow's `OFPAT_COPY_TTL_OUT`.
- `of_copy_ttl_in`: OpenFlow's `OFPAT_COPY_TTL_IN`.
- `of_pop_vlan`: OpenFlow's `OFPAT_POP_VLAN`.
- `of_push_vlan`: OpenFlow's `OFPAT_PUSH_VLAN`.
  - `ethertype`: Ethertype.
- `of_set_vlan_vid`: OpenFlow's `OFPAT_SET_VLAN_VID`.
  - `vlan_vid`: VLAN id.
- `of_set_vlan_pcp`: OpenFlow's `OFPAT_SET_VLAN_PCP`.
  - `vlan_pcp`: VLAN priority.
- `of_pop_mpls`: OpenFlow's `OFPAT_POP_MPLS`.

  - `ethertype`: Ethertype.

- `of_push_mpls`: OpenFlow's `OFPAT_PUSH_MPLS`.

  - `ethertype`: Ethertype.

- `vxlan_encap`: Performs a VXLAN encapsulation, outer layer configuration is done through *Config VXLAN Encap outer layers*.

- `vxlan_decap`: Performs a decapsulation action by stripping all headers of the VXLAN tunnel network overlay from the matched flow.

- `nvgre_encap`: Performs a NVGRE encapsulation, outer layer configuration is done through *Config NVGRE Encap outer layers*.

- `nvgre_decap`: Performs a decapsulation action by stripping all headers of the NVGRE tunnel network overlay from the matched flow.

- `l2_encap`: Performs a L2 encapsulation, L2 configuration is done through *Config L2 Encap*.

- `l2_decap`: Performs a L2 decapsulation, L2 configuration is done through *Config L2 Decap*.

- `mplsogre_encap`: Performs a MPLSoGRE encapsulation, outer layer configuration is done through *Config MPLSoGRE Encap outer layers*.

- `mplsogre_decap`: Performs a MPLSoGRE decapsulation, outer layer configuration is done through *Config MPLSoGRE Decap outer layers*.

- `mplsoudp_encap`: Performs a MPLSoUDP encapsulation, outer layer configuration is done through *Config MPLSoUDP Encap outer layers*.

- `mplsoudp_decap`: Performs a MPLSoUDP decapsulation, outer layer configuration is done through *Config MPLSoUDP Decap outer layers*.

- `set_ipv4_src`: Set a new IPv4 source address in the outermost IPv4 header.

  - `ipv4_addr`: New IPv4 source address.

- `set_ipv4_dst`: Set a new IPv4 destination address in the outermost IPv4 header.

  - `ipv4_addr`: New IPv4 destination address.

- `set_ipv6_src`: Set a new IPv6 source address in the outermost IPv6 header.

  - `ipv6_addr`: New IPv6 source address.

- `set_ipv6_dst`: Set a new IPv6 destination address in the outermost IPv6 header.

  - `ipv6_addr`: New IPv6 destination address.

- `set_tp_src`: Set a new source port number in the outermost TCP/UDP header.

  - `port`: New TCP/UDP source port number.

- `set_tp_dst`: Set a new destination port number in the outermost TCP/UDP header.

  - `port`: New TCP/UDP destination port number.

- `mac_swap`: Swap the source and destination MAC addresses in the outermost Ethernet header.

- `dec_ttl`: Performs a decrease TTL value action

- `set_ttl`: Set TTL value with specified value - `ttl_value {unsigned}`: The new TTL value to be set

- `set_mac_src`: set source MAC address
  - `mac_addr {MAC-48}`: new source MAC address
- `set_mac_dst`: set destination MAC address
  - `mac_addr {MAC-48}`: new destination MAC address
- `inc_tcp_seq`: Increase sequence number in the outermost TCP header.
  - `value {unsigned}`: Value to increase TCP sequence number by.
- `dec_tcp_seq`: Decrease sequence number in the outermost TCP header.
  - `value {unsigned}`: Value to decrease TCP sequence number by.
- `inc_tcp_ack`: Increase acknowledgment number in the outermost TCP header.
  - `value {unsigned}`: Value to increase TCP acknowledgment number by.
- `dec_tcp_ack`: Decrease acknowledgment number in the outermost TCP header.
  - `value {unsigned}`: Value to decrease TCP acknowledgment number by.
- `set_ipv4_dscp`: Set IPv4 DSCP value with specified value
  - `dscp_value {unsigned}`: The new DSCP value to be set
- `set_ipv6_dscp`: Set IPv6 DSCP value with specified value
  - `dscp_value {unsigned}`: The new DSCP value to be set

Destroying flow rules

`flow destroy` destroys one or more rules from their rule ID (as returned by `flow create`), this command calls `rte_flow_destroy()` as many times as necessary:

```
flow destroy {port_id} rule {rule_id} [...]
```

If successful, it will show:

```
Flow rule #[...] destroyed
```

It does not report anything for rule IDs that do not exist. The usual error message is shown when a rule cannot be destroyed:

```
Caught error type [...] ([...]): [...]
```

`flow flush` destroys all rules on a device and does not take extra arguments. It is bound to `rte_flow_flush()`:

```
flow flush {port_id}
```

Any errors are reported as above.

Creating several rules and destroying them:

```
testpmd> flow create 0 ingress pattern eth / ipv6 / end
   actions queue index 2 / end
Flow rule #0 created
```

(continues on next page)

(continued from previous page)

```
testpmd> flow create 0 ingress pattern eth / ipv4 / end
   actions queue index 3 / end
Flow rule #1 created
testpmd> flow destroy 0 rule 0 rule 1
Flow rule #1 destroyed
Flow rule #0 destroyed
testpmd>
```

The same result can be achieved using `flow flush`:

```
testpmd> flow create 0 ingress pattern eth / ipv6 / end
   actions queue index 2 / end
Flow rule #0 created
testpmd> flow create 0 ingress pattern eth / ipv4 / end
   actions queue index 3 / end
Flow rule #1 created
testpmd> flow flush 0
testpmd>
```

Non-existent rule IDs are ignored:

```
testpmd> flow create 0 ingress pattern eth / ipv6 / end
   actions queue index 2 / end
Flow rule #0 created
testpmd> flow create 0 ingress pattern eth / ipv4 / end
   actions queue index 3 / end
Flow rule #1 created
testpmd> flow destroy 0 rule 42 rule 10 rule 2
testpmd>
testpmd> flow destroy 0 rule 0
Flow rule #0 destroyed
testpmd>
```

Querying flow rules

`flow query` queries a specific action of a flow rule having that ability. Such actions collect information that can be reported using this command. It is bound to `rte_flow_query()`:

```
flow query {port_id} {rule_id} {action}
```

If successful, it will display either the retrieved data for known actions or the following message:

```
Cannot display result for action type [...] ([...])
```

Otherwise, it will complain either that the rule does not exist or that some error occurred:

```
Flow rule #[...] not found
```

```
Caught error type [...] ([...]): [...]
```

Currently only the `count` action is supported. This action reports the number of packets that hit the flow rule and the total number of bytes. Its output has the following format:

```
count:
 hits_set: [...] # whether "hits" contains a valid value
```

(continues on next page)

(continued from previous page)

```
 bytes_set: [...] # whether "bytes" contains a valid value
 hits: [...] # number of packets
 bytes: [...] # number of bytes
```

Querying counters for TCPv6 packets redirected to queue 6:

```
testpmd> flow create 0 ingress pattern eth / ipv6 / tcp / end
   actions queue index 6 / count / end
Flow rule #4 created
testpmd> flow query 0 4 count
count:
 hits_set: 1
 bytes_set: 0
 hits: 386446
 bytes: 0
testpmd>
```

Listing flow rules

`flow list` lists existing flow rules sorted by priority and optionally filtered by group identifiers:

```
flow list {port_id} [group {group_id}] [...]
```

This command only fails with the following message if the device does not exist:

```
Invalid port [...]
```

Output consists of a header line followed by a short description of each flow rule, one per line. There is no output at all when no flow rules are configured on the device:

```
ID      Group   Prio    Attr    Rule
[...]   [...]   [...]   [...]   [...]
```

`Attr` column flags:

- `i` for `ingress`.
- `e` for `egress`.

Creating several flow rules and listing them:

```
testpmd> flow create 0 ingress pattern eth / ipv4 / end
   actions queue index 6 / end
Flow rule #0 created
testpmd> flow create 0 ingress pattern eth / ipv6 / end
   actions queue index 2 / end
Flow rule #1 created
testpmd> flow create 0 priority 5 ingress pattern eth / ipv4 / udp / end
   actions rss queues 6 7 8 end / end
Flow rule #2 created
testpmd> flow list 0
ID      Group   Prio    Attr    Rule
0       0       0       i-      ETH IPV4 => QUEUE
1       0       0       i-      ETH IPV6 => QUEUE
2       0       5       i-      ETH IPV4 UDP => RSS
testpmd>
```

Rules are sorted by priority (i.e. group ID first, then priority level):

```
testpmd> flow list 1
ID      Group   Prio    Attr    Rule
0       0       0       i-      ETH => COUNT
6       0       500     i-      ETH IPV6 TCP => DROP COUNT
5       0       1000    i-      ETH IPV6 ICMP => QUEUE
1       24      0       i-      ETH IPV4 UDP => QUEUE
4       24      10      i-      ETH IPV4 TCP => DROP
3       24      20      i-      ETH IPV4 => DROP
2       24      42      i-      ETH IPV4 UDP => QUEUE
7       63      0       i-      ETH IPV6 UDP VXLAN => MARK QUEUE
testpmd>
```

Output can be limited to specific groups:

```
testpmd> flow list 1 group 0 group 63
ID      Group   Prio    Attr    Rule
0       0       0       i-      ETH => COUNT
6       0       500     i-      ETH IPV6 TCP => DROP COUNT
5       0       1000    i-      ETH IPV6 ICMP => QUEUE
7       63      0       i-      ETH IPV6 UDP VXLAN => MARK QUEUE
testpmd>
```

Toggling isolated mode

`flow isolate` can be used to tell the underlying PMD that ingress traffic must only be injected from the defined flow rules; that no default traffic is expected outside those rules and the driver is free to assign more resources to handle them. It is bound to `rte_flow_isolate()`:

```
flow isolate {port_id} {boolean}
```

If successful, enabling or disabling isolated mode shows either:

```
Ingress traffic on port [...]
   is now restricted to the defined flow rules
```

Or:

```
Ingress traffic on port [...]
   is not restricted anymore to the defined flow rules
```

Otherwise, in case of error:

```
Caught error type [...] ([...]): [...]
```

Mainly due to its side effects, PMDs supporting this mode may not have the ability to toggle it more than once without reinitializing affected ports first (e.g. by exiting testpmd).

Enabling isolated mode:

```
testpmd> flow isolate 0 true
Ingress traffic on port 0 is now restricted to the defined flow rules
testpmd>
```

Disabling isolated mode:

```
testpmd> flow isolate 0 false
Ingress traffic on port 0 is not restricted anymore to the defined flow rules
testpmd>
```

Dumping HW internal information

`flow dump` dumps the hardware's internal representation information of all flows. It is bound to `rte_flow_dev_dump()`:

```
flow dump {port_id} {output_file}
```

If successful, it will show:

```
Flow dump finished
```

Otherwise, it will complain error occurred:

```
Caught error type [...] ([...]): [...]
```

Listing and destroying aged flow rules

`flow aged` simply lists aged flow rules be get from api `rte_flow_get_aged_flows`, and `destroy` parameter can be used to destroy those flow rules in PMD.

flow aged {port_id} [destroy]

Listing current aged flow rules:

```
testpmd> flow aged 0
Port 0 total aged flows: 0
testpmd> flow create 0 ingress pattern eth / ipv4 src is 2.2.2.14 / end
   actions age timeout 5 / queue index 0 /  end
Flow rule #0 created
testpmd> flow create 0 ingress pattern eth / ipv4 src is 2.2.2.15 / end
   actions age timeout 4 / queue index 0 /  end
Flow rule #1 created
testpmd> flow create 0 ingress pattern eth / ipv4 src is 2.2.2.16 / end
   actions age timeout 2 / queue index 0 /  end
Flow rule #2 created
testpmd> flow create 0 ingress pattern eth / ipv4 src is 2.2.2.17 / end
   actions age timeout 3 / queue index 0 /  end
Flow rule #3 created
```

Aged Rules are simply list as command `flow list {port_id}`, but strip the detail rule information, all the aged flows are sorted by the longest timeout time. For example, if those rules be configured in the same time, ID 2 will be the first aged out rule, the next will be ID 3, ID 1, ID 0:

```
testpmd> flow aged 0
Port 0 total aged flows: 4
ID      Group   Prio    Attr
2       0       0       i--
3       0       0       i--
1       0       0       i--
0       0       0       i--
```

If attach `destroy` parameter, the command will destroy all the list aged flow rules.

testpmd> flow aged 0 destroy Port 0 total aged flows: 4 ID Group Prio Attr 2 0 0 i– 3 0 0 i– 1 0 0 i– 0 0 0 i–

Flow rule #2 destroyed Flow rule #3 destroyed Flow rule #1 destroyed Flow rule #0 destroyed 4 flows be destroyed testpmd> flow aged 0 Port 0 total aged flows: 0

Sample QinQ flow rules

Before creating QinQ rule(s) the following commands should be issued to enable QinQ:

```
testpmd> port stop 0
testpmd> vlan set qinq_strip on 0
```

The above command sets the inner and outer TPID's to 0x8100.

To change the TPID's the following commands should be used:

```
testpmd> vlan set outer tpid 0xa100 0
testpmd> vlan set inner tpid 0x9100 0
testpmd> port start 0
```

Validate and create a QinQ rule on port 0 to steer traffic to a VF queue in a VM.

```
testpmd> flow validate 0 ingress pattern eth / vlan tci is 123 /
    vlan tci is 456 / end actions vf id 1 / queue index 0 / end
Flow rule #0 validated

testpmd> flow create 0 ingress pattern eth / vlan tci is 4 /
    vlan tci is 456 / end actions vf id 123 / queue index 0 / end
Flow rule #0 created

testpmd> flow list 0
ID      Group   Prio    Attr    Rule
0       0       0       i-      ETH VLAN VLAN=>VF QUEUE
```

Validate and create a QinQ rule on port 0 to steer traffic to a queue on the host.

```
testpmd> flow validate 0 ingress pattern eth / vlan tci is 321 /
     vlan tci is 654 / end actions pf / queue index 0 / end
Flow rule #1 validated

testpmd> flow create 0 ingress pattern eth / vlan tci is 321 /
     vlan tci is 654 / end actions pf / queue index 1 / end
Flow rule #1 created

testpmd> flow list 0
ID      Group   Prio    Attr    Rule
0       0       0       i-      ETH VLAN VLAN=>VF QUEUE
1       0       0       i-      ETH VLAN VLAN=>PF QUEUE
```

Sample VXLAN encapsulation rule

VXLAN encapsulation outer layer has default value pre-configured in testpmd source code, those can be changed by using the following commands

IPv4 VXLAN outer header:

```
testpmd> set vxlan ip-version ipv4 vni 4 udp-src 4 udp-dst 4 ip-src 127.0.0.1
       ip-dst 128.0.0.1 eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern end actions vxlan_encap /
       queue index 0 / end

testpmd> set vxlan-with-vlan ip-version ipv4 vni 4 udp-src 4 udp-dst 4 ip-src
        127.0.0.1 ip-dst 128.0.0.1 vlan-tci 34 eth-src 11:11:11:11:11:11
        eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern end actions vxlan_encap /
        queue index 0 / end

testpmd> set vxlan-tos-ttl ip-version ipv4 vni 4 udp-src 4 udp-dst 4 ip-tos 0
        ip-ttl 255 ip-src 127.0.0.1 ip-dst 128.0.0.1 eth-src 11:11:11:11:11:11
        eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern end actions vxlan_encap /
        queue index 0 / end
```

IPv6 VXLAN outer header:

```
testpmd> set vxlan ip-version ipv6 vni 4 udp-src 4 udp-dst 4 ip-src ::1
       ip-dst ::2222 eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern end actions vxlan_encap /
        queue index 0 / end

testpmd> set vxlan-with-vlan ip-version ipv6 vni 4 udp-src 4 udp-dst 4
        ip-src ::1 ip-dst ::2222 vlan-tci 34 eth-src 11:11:11:11:11:11
        eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern end actions vxlan_encap /
        queue index 0 / end

testpmd> set vxlan-tos-ttl ip-version ipv6 vni 4 udp-src 4 udp-dst 4
        ip-tos 0 ip-ttl 255 ::1 ip-dst ::2222 eth-src 11:11:11:11:11:11
        eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern end actions vxlan_encap /
        queue index 0 / end
```

Sample NVGRE encapsulation rule

NVGRE encapsulation outer layer has default value pre-configured in testpmd source code, those can be changed by using the following commands

IPv4 NVGRE outer header:

```
testpmd> set nvgre ip-version ipv4 tni 4 ip-src 127.0.0.1 ip-dst 128.0.0.1
       eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern end actions nvgre_encap /
       queue index 0 / end

testpmd> set nvgre-with-vlan ip-version ipv4 tni 4 ip-src 127.0.0.1
        ip-dst 128.0.0.1 vlan-tci 34 eth-src 11:11:11:11:11:11
        eth-dst 22:22:22:22:22:22
```

(continues on next page)

(continued from previous page)

```
testpmd> flow create 0 ingress pattern end actions nvgre_encap /
        queue index 0 / end
```

IPv6 NVGRE outer header:

```
testpmd> set nvgre ip-version ipv6 tni 4 ip-src ::1 ip-dst ::2222
       eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern end actions nvgre_encap /
       queue index 0 / end

testpmd> set nvgre-with-vlan ip-version ipv6 tni 4 ip-src ::1 ip-dst ::2222
       vlan-tci 34 eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern end actions nvgre_encap /
       queue index 0 / end
```

Sample L2 encapsulation rule

L2 encapsulation has default value pre-configured in testpmd source code, those can be changed by using the following commands

L2 header:

```
testpmd> set l2_encap ip-version ipv4
       eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / mpls / end actions
       mplsoudp_decap / l2_encap / end
```

L2 with VXLAN header:

```
testpmd> set l2_encap-with-vlan ip-version ipv4 vlan-tci 34
        eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / mpls / end actions
       mplsoudp_decap / l2_encap / end
```

Sample L2 decapsulation rule

L2 decapsulation has default value pre-configured in testpmd source code, those can be changed by using the following commands

L2 header:

```
testpmd> set l2_decap
testpmd> flow create 0 egress pattern eth / end actions l2_decap / mplsoudp_encap /
       queue index 0 / end
```

L2 with VXLAN header:

```
testpmd> set l2_encap-with-vlan
testpmd> flow create 0 egress pattern eth / end actions l2_encap / mplsoudp_encap /
        queue index 0 / end
```

Sample MPLSoGRE encapsulation rule

MPLSoGRE encapsulation outer layer has default value pre-configured in testpmd source code, those can be changed by using the following commands

IPv4 MPLSoGRE outer header:

```
testpmd> set mplsogre_encap ip-version ipv4 label 4
       ip-src 127.0.0.1 ip-dst 128.0.0.1 eth-src 11:11:11:11:11:11
       eth-dst 22:22:22:22:22:22
testpmd> flow create 0 egress pattern eth / end actions l2_decap /
       mplsogre_encap / end
```

IPv4 MPLSoGRE with VLAN outer header:

```
testpmd> set mplsogre_encap-with-vlan ip-version ipv4 label 4
       ip-src 127.0.0.1 ip-dst 128.0.0.1 vlan-tci 34
       eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 egress pattern eth / end actions l2_decap /
       mplsogre_encap / end
```

IPv6 MPLSoGRE outer header:

```
testpmd> set mplsogre_encap ip-version ipv6 mask 4
       ip-src ::1 ip-dst ::2222 eth-src 11:11:11:11:11:11
       eth-dst 22:22:22:22:22:22
testpmd> flow create 0 egress pattern eth / end actions l2_decap /
       mplsogre_encap / end
```

IPv6 MPLSoGRE with VLAN outer header:

```
testpmd> set mplsogre_encap-with-vlan ip-version ipv6 mask 4
       ip-src ::1 ip-dst ::2222 vlan-tci 34
       eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 egress pattern eth / end actions l2_decap /
       mplsogre_encap / end
```

Sample MPLSoGRE decapsulation rule

MPLSoGRE decapsulation outer layer has default value pre-configured in testpmd source code, those can be changed by using the following commands

IPv4 MPLSoGRE outer header:

```
testpmd> set mplsogre_decap ip-version ipv4
testpmd> flow create 0 ingress pattern eth / ipv4 / gre / mpls / end actions
       mplsogre_decap / l2_encap / end
```

IPv4 MPLSoGRE with VLAN outer header:

```
testpmd> set mplsogre_decap-with-vlan ip-version ipv4
testpmd> flow create 0 ingress pattern eth / vlan / ipv4 / gre / mpls / end
       actions mplsogre_decap / l2_encap / end
```

IPv6 MPLSoGRE outer header:

```
testpmd> set mplsogre_decap ip-version ipv6
testpmd> flow create 0 ingress pattern eth / ipv6 / gre / mpls / end
       actions mplsogre_decap / l2_encap / end
```

IPv6 MPLSoGRE with VLAN outer header:

```
testpmd> set mplsogre_decap-with-vlan ip-version ipv6
testpmd> flow create 0 ingress pattern eth / vlan / ipv6 / gre / mpls / end
       actions mplsogre_decap / l2_encap / end
```

Sample MPLSoUDP encapsulation rule

MPLSoUDP encapsulation outer layer has default value pre-configured in testpmd source code, those can be changed by using the following commands

IPv4 MPLSoUDP outer header:

```
testpmd> set mplsoudp_encap ip-version ipv4 label 4 udp-src 5 udp-dst 10
       ip-src 127.0.0.1 ip-dst 128.0.0.1 eth-src 11:11:11:11:11:11
       eth-dst 22:22:22:22:22:22
testpmd> flow create 0 egress pattern eth / end actions l2_decap /
       mplsoudp_encap / end
```

IPv4 MPLSoUDP with VLAN outer header:

```
testpmd> set mplsoudp_encap-with-vlan ip-version ipv4 label 4 udp-src 5
       udp-dst 10 ip-src 127.0.0.1 ip-dst 128.0.0.1 vlan-tci 34
       eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 egress pattern eth / end actions l2_decap /
       mplsoudp_encap / end
```

IPv6 MPLSoUDP outer header:

```
testpmd> set mplsoudp_encap ip-version ipv6 mask 4 udp-src 5 udp-dst 10
       ip-src ::1 ip-dst ::2222 eth-src 11:11:11:11:11:11
       eth-dst 22:22:22:22:22:22
testpmd> flow create 0 egress pattern eth / end actions l2_decap /
       mplsoudp_encap / end
```

IPv6 MPLSoUDP with VLAN outer header:

```
testpmd> set mplsoudp_encap-with-vlan ip-version ipv6 mask 4 udp-src 5
       udp-dst 10 ip-src ::1 ip-dst ::2222 vlan-tci 34
       eth-src 11:11:11:11:11:11 eth-dst 22:22:22:22:22:22
testpmd> flow create 0 egress pattern eth / end actions l2_decap /
       mplsoudp_encap / end
```

Sample MPLSoUDP decapsulation rule

MPLSoUDP decapsulation outer layer has default value pre-configured in testpmd source code, those can be changed by using the following commands

IPv4 MPLSoUDP outer header:

```
testpmd> set mplsoudp_decap ip-version ipv4
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / mpls / end actions
       mplsoudp_decap / l2_encap / end
```

IPv4 MPLSoUDP with VLAN outer header:

```
testpmd> set mplsoudp_decap-with-vlan ip-version ipv4
testpmd> flow create 0 ingress pattern eth / vlan / ipv4 / udp / mpls / end
       actions mplsoudp_decap / l2_encap / end
```

IPv6 MPLSoUDP outer header:

```
testpmd> set mplsoudp_decap ip-version ipv6
testpmd> flow create 0 ingress pattern eth / ipv6 / udp / mpls / end
       actions mplsoudp_decap / l2_encap / end
```

IPv6 MPLSoUDP with VLAN outer header:

```
testpmd> set mplsoudp_decap-with-vlan ip-version ipv6
testpmd> flow create 0 ingress pattern eth / vlan / ipv6 / udp / mpls / end
       actions mplsoudp_decap / l2_encap / end
```

Sample Raw encapsulation rule

Raw encapsulation configuration can be set by the following commands

Eecapsulating VxLAN:

```
testpmd> set raw_encap 4 eth src is 10:11:22:33:44:55 / vlan tci is 1
       inner_type is 0x0800 / ipv4 / udp dst is 4789 / vxlan vni
       is 2 / end_set
testpmd> flow create 0 egress pattern eth / ipv4 / end actions
       raw_encap index 4 / end
```

Sample Raw decapsulation rule

Raw decapsulation configuration can be set by the following commands

Decapsulating VxLAN:

```
testpmd> set raw_decap eth / ipv4 / udp / vxlan / end_set
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / vxlan / eth / ipv4 /
       end actions raw_decap / queue index 0 / end
```

Sample ESP rules

ESP rules can be created by the following commands:

```
testpmd> flow create 0 ingress pattern eth / ipv4 / esp spi is 1 / end actions
       queue index 3 / end
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / esp spi is 1 / end
       actions queue index 3 / end
testpmd> flow create 0 ingress pattern eth / ipv6 / esp spi is 1 / end actions
       queue index 3 / end
testpmd> flow create 0 ingress pattern eth / ipv6 / udp / esp spi is 1 / end
       actions queue index 3 / end
```

Sample AH rules

AH rules can be created by the following commands:

```
testpmd> flow create 0 ingress pattern eth / ipv4 / ah spi is 1 / end actions
       queue index 3 / end
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / ah spi is 1 / end
       actions queue index 3 / end
testpmd> flow create 0 ingress pattern eth / ipv6 / ah spi is 1 / end actions
       queue index 3 / end
testpmd> flow create 0 ingress pattern eth / ipv6 / udp / ah spi is 1 / end
       actions queue index 3 / end
```

Sample PFCP rules

PFCP rules can be created by the following commands(s_field need to be 1 if seid is set):

```
testpmd> flow create 0 ingress pattern eth / ipv4 / pfcp s_field is 0 / end
       actions queue index 3 / end
testpmd> flow create 0 ingress pattern eth / ipv4 / pfcp s_field is 1
       seid is 1 / end actions queue index 3 / end
testpmd> flow create 0 ingress pattern eth / ipv6 / pfcp s_field is 0 / end
       actions queue index 3 / end
testpmd> flow create 0 ingress pattern eth / ipv6 / pfcp s_field is 1
       seid is 1 / end actions queue index 3 / end
```

8.4.14 BPF Functions

The following sections show functions to load/unload eBPF based filters.

bpf-load

Load an eBPF program as a callback for particular RX/TX queue:

```
testpmd> bpf-load rx|tx (portid) (queueid) (load-flags) (bpf-prog-filename)
```

The available load-flags are:

- `J`: use JIT generated native code, otherwise BPF interpreter will be used.

- `M`: assume input parameter is a pointer to rte_mbuf, otherwise assume it is a pointer to first segment's data.
- `-`: none.

Note: You'll need clang v3.7 or above to build bpf program you'd like to load

For example:

```
cd examples/bpf
clang -O2 -target bpf -c t1.c
```

Then to load (and JIT compile) t1.o at RX queue 0, port 1:

```
testpmd> bpf-load rx 1 0 J ./dpdk.org/examples/bpf/t1.o
```

To load (not JITed) t1.o at TX queue 0, port 0:

```
testpmd> bpf-load tx 0 0 - ./dpdk.org/examples/bpf/t1.o
```

bpf-unload

Unload previously loaded eBPF program for particular RX/TX queue:

```
testpmd> bpf-unload rx|tx (portid) (queueid)
```

For example to unload BPF filter from TX queue 0, port 0:

```
testpmd> bpf-unload tx 0 0
```

CHAPTER

NINE

NETWORK INTERFACE CONTROLLER DRIVERS

9.1 Overview of Networking Drivers

The networking drivers may be classified in two categories:

- physical for real devices
- virtual for emulated devices

Some physical devices may be shaped through a virtual layer as for SR-IOV. The interface seen in the virtual environment is a VF (Virtual Function).

The ethdev layer exposes an API to use the networking functions of these devices. The bottom half part of ethdev is implemented by the drivers. Thus some features may not be implemented.

There are more differences between drivers regarding some internal properties, portability or even documentation availability. Most of these differences are summarized below.

More details about features can be found in *Features Overview*.

Feature	af_xdp	afpacket	ark	atlantic	avp	axgbe	bnx2x	br
Speed capabilities			Y	Y		Y	P	Y
Link status	Y			Y	Y	Y	Y	Y
Link status event				Y			Y	Y
Removal event								
Queue status event								
Rx interrupt								Y
Lock-free Tx queue								
Fast mbuf free								
Free Tx mbuf on demand								
Queue start/stop			Y	Y				Y
Runtime Rx queue setup								
Runtime Tx queue setup								
Burst mode info								
MTU update	Y			Y				Y
Jumbo frame			Y	Y	Y	Y		Y
Scattered Rx			Y		Y	Y		Y
LRO								Y
TSO								Y
Promiscuous mode	Y			Y	Y	Y	Y	Y

Feature	af_xdp	afpacket	ark	atlantic	avp	axgbe	bnx2x	br
Allmulticast mode				Y		Y		Y
Unicast MAC filter				Y	Y		Y	Y
Multicast MAC filter							Y	Y
RSS hash				Y		Y		Y
RSS key update				Y				Y
RSS reta update				Y				Y
Inner RSS								
VMDq								Y
SR-IOV							Y	Y
DCB								
VLAN filter				Y				Y
Flow control				Y				Y
Flow API								Y
Rate limitation								
Traffic mirroring								
Inline crypto								
Inline protocol								
CRC offload				Y		Y		Y
VLAN offload				Y	Y			Y
QinQ offload								
L3 checksum offload				Y		Y		Y
L4 checksum offload				Y		Y		Y
Timestamp offload								
MACsec offload				Y				
Inner L3 checksum								Y
Inner L4 checksum								Y
Packet type parsing				Y				Y
Timesync								Y
Rx descriptor status				Y				Y
Tx descriptor status				Y				Y
Basic stats			Y	Y	Y	Y	Y	Y
Extended stats				Y			Y	Y
Stats per queue	Y		Y	Y	Y			Y
FW version				Y				Y
EEPROM dump				Y				Y
Module EEPROM dump								
Registers dump				Y				
LED								Y
Multiprocess aware								Y
BSD nic_uio								
Linux UIO			Y	Y	Y	Y	Y	Y
Linux VFIO								Y
Other kdrv								
ARMv7								
ARMv8				Y				Y
Power8								
x86-32				Y		Y		Y

Feature	af_xdp	afpacket	ark	atlantic	avp	axgbe	bnx2x	br
x86-64	Y		Y	Y	Y	Y	Y	Y
Usage doc			Y				Y	
Design doc								
Perf doc								

Note: Features marked with "P" are partially supported. Refer to the appropriate NIC guide in the following sections for details.

9.2 Features Overview

This section explains the supported features that are listed in the *Overview of Networking Drivers*.

As a guide to implementers it also shows the structs where the features are defined and the APIs that can be use to get/set the values.

Following tags used for feature details, these are from driver point of view:

`[uses]` : Driver uses some kind of input from the application.

`[implements]` : Driver implements a functionality.

`[provides]` : Driver provides some kind of data to the application. It is possible to provide data by implementing some function, but "provides" is used for cases where provided data can't be represented simply by a function.

`[related]` : Related API with that feature.

9.2.1 Speed capabilities

Supports getting the speed capabilities that the current device is capable of.

- **[provides] rte_eth_dev_info**: `speed_capa:ETH_LINK_SPEED_*`.
- **[related] API**: `rte_eth_dev_info_get()`.

9.2.2 Link status

Supports getting the link speed, duplex mode and link state (up/down).

- **[implements] eth_dev_ops**: `link_update`.
- **[implements] rte_eth_dev_data**: `dev_link`.
- **[related] API**: `rte_eth_link_get()`, `rte_eth_link_get_nowait()`.

9.2.3 Link status event

Supports Link Status Change interrupts.

- **[uses] user config**: `dev_conf.intr_conf.lsc`.
- **[uses] rte_eth_dev_data**: `dev_flags:RTE_ETH_DEV_INTR_LSC`.
- **[uses] rte_eth_event_type**: `RTE_ETH_EVENT_INTR_LSC`.
- **[implements] rte_eth_dev_data**: `dev_link`.
- **[provides] rte_pci_driver.drv_flags**: `RTE_PCI_DRV_INTR_LSC`.
- **[related] API**: `rte_eth_link_get()`, `rte_eth_link_get_nowait()`.

9.2.4 Removal event

Supports device removal interrupts.

- **[uses] user config**: `dev_conf.intr_conf.rmv`.
- **[uses] rte_eth_dev_data**: `dev_flags:RTE_ETH_DEV_INTR_RMV`.
- **[uses] rte_eth_event_type**: `RTE_ETH_EVENT_INTR_RMV`.
- **[provides] rte_pci_driver.drv_flags**: `RTE_PCI_DRV_INTR_RMV`.

9.2.5 Queue status event

Supports queue enable/disable events.

- **[uses] rte_eth_event_type**: `RTE_ETH_EVENT_QUEUE_STATE`.

9.2.6 Rx interrupt

Supports Rx interrupts.

- **[uses] user config**: `dev_conf.intr_conf.rxq`.
- **[implements] eth_dev_ops**: `rx_queue_intr_enable`, `rx_queue_intr_disable`.
- **[related] API**: `rte_eth_dev_rx_intr_enable()`, `rte_eth_dev_rx_intr_disable()`.

9.2.7 Lock-free Tx queue

If a PMD advertises DEV_TX_OFFLOAD_MT_LOCKFREE capable, multiple threads can invoke rte_eth_tx_burst() concurrently on the same Tx queue without SW lock.

- **[uses] rte_eth_txconf,rte_eth_txmode**: `offloads:DEV_TX_OFFLOAD_MT_LOCKFREE`.
- **[provides] rte_eth_dev_info**: `tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_MT_LOCKFREE`
- **[related] API**: `rte_eth_tx_burst()`.

9.2.8 Fast mbuf free

Supports optimization for fast release of mbufs following successful Tx. Requires that per queue, all mbufs come from the same mempool and has refcnt = 1.

- **[uses] rte_eth_txconf,rte_eth_txmode**: `offloads:DEV_TX_OFFLOAD_MBUF_FAST_FREE`.
- **[provides] rte_eth_dev_info**: `tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_MBUF_FAST_F`

9.2.9 Free Tx mbuf on demand

Supports freeing consumed buffers on a Tx ring.

- **[implements] eth_dev_ops**: `tx_done_cleanup`.
- **[related] API**: `rte_eth_tx_done_cleanup()`.

9.2.10 Queue start/stop

Supports starting/stopping a specific Rx/Tx queue of a port.

- **[implements] eth_dev_ops**: `rx_queue_start`, `rx_queue_stop`, `tx_queue_start`, `tx_queue_stop`.
- **[related] API**: `rte_eth_dev_rx_queue_start()`, `rte_eth_dev_rx_queue_stop()`, `rte_eth_dev_tx_queue_start()`, `rte_eth_dev_tx_queue_stop()`.

9.2.11 MTU update

Supports updating port MTU.

- **[implements] eth_dev_ops**: `mtu_set`.
- **[implements] rte_eth_dev_data**: `mtu`.
- **[provides] rte_eth_dev_info**: `max_rx_pktlen`.
- **[related] API**: `rte_eth_dev_set_mtu()`, `rte_eth_dev_get_mtu()`.

9.2.12 Jumbo frame

Supports Rx jumbo frames.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_JUMBO_FRAME`. `dev_conf.rxmode.max_rx_pkt_len`.
- **[related] rte_eth_dev_info**: `max_rx_pktlen`.
- **[related] API**: `rte_eth_dev_set_mtu()`.

9.2.13 Scattered Rx

Supports receiving segmented mbufs.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_SCATTER`.
- **[implements] datapath**: `Scattered Rx function`.
- **[implements] rte_eth_dev_data**: `scattered_rx`.
- **[provides] eth_dev_ops**: `rxq_info_get:scattered_rx`.
- **[related] eth_dev_ops**: `rx_pkt_burst`.

9.2.14 LRO

Supports Large Receive Offload.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_TCP_LRO`. `dev_conf.rxmode.max_lro_pkt_size`.
- **[implements] datapath**: `LRO functionality`.
- **[implements] rte_eth_dev_data**: `lro`.
- **[provides] mbuf**: `mbuf.ol_flags:PKT_RX_LRO`, `mbuf.tso_segsz`.
- **[provides] rte_eth_dev_info**: `rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_TCP_LRO`.
- **[provides] rte_eth_dev_info**: `max_lro_pkt_size`.

9.2.15 TSO

Supports TCP Segmentation Offloading.

- **[uses] rte_eth_txconf,rte_eth_txmode**: `offloads:DEV_TX_OFFLOAD_TCP_TSO`.
- **[uses] rte_eth_desc_lim**: `nb_seg_max`, `nb_mtu_seg_max`.
- **[uses] mbuf**: `mbuf.ol_flags:` `PKT_TX_TCP_SEG`, `PKT_TX_IPV4`, `PKT_TX_IPV6`, `PKT_TX_IP_CKSUM`.
- **[uses] mbuf**: `mbuf.tso_segsz`, `mbuf.l2_len`, `mbuf.l3_len`, `mbuf.l4_len`.
- **[implements] datapath**: `TSO functionality`.
- **[provides] rte_eth_dev_info**: `tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_TCP_TSO`, `DEV_TX_OFFLOAD_UDP_TSO`.

9.2.16 Promiscuous mode

Supports enabling/disabling promiscuous mode for a port.

- **[implements] eth_dev_ops**: `promiscuous_enable`, `promiscuous_disable`.
- **[implements] rte_eth_dev_data**: `promiscuous`.
- **[related] API**: `rte_eth_promiscuous_enable()`, `rte_eth_promiscuous_disable()`, `rte_eth_promiscuous_get()`.

9.2.17 Allmulticast mode

Supports enabling/disabling receiving multicast frames.

- **[implements] eth_dev_ops**: `allmulticast_enable`, `allmulticast_disable`.
- **[implements] rte_eth_dev_data**: `all_multicast`.
- **[related] API**: `rte_eth_allmulticast_enable()`, `rte_eth_allmulticast_disable()`, `rte_eth_allmulticast_get()`.

9.2.18 Unicast MAC filter

Supports adding MAC addresses to enable whitelist filtering to accept packets.

- **[implements] eth_dev_ops**: `mac_addr_set`, `mac_addr_add`, `mac_addr_remove`.
- **[implements] rte_eth_dev_data**: `mac_addrs`.
- **[related] API**: `rte_eth_dev_default_mac_addr_set()`, `rte_eth_dev_mac_addr_add()`, `rte_eth_dev_mac_addr_remove()`, `rte_eth_macaddr_get()`.

9.2.19 Multicast MAC filter

Supports setting multicast addresses to filter.

- **[implements] eth_dev_ops**: `set_mc_addr_list`.
- **[related] API**: `rte_eth_dev_set_mc_addr_list()`.

9.2.20 RSS hash

Supports RSS hashing on RX.

- **[uses] user config**: `dev_conf.rxmode.mq_mode` = `ETH_MQ_RX_RSS_FLAG`.
- **[uses] user config**: `dev_conf.rx_adv_conf.rss_conf`.
- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_RSS_HASH`.
- **[provides] rte_eth_dev_info**: `flow_type_rss_offloads`.
- **[provides] mbuf**: `mbuf.ol_flags:PKT_RX_RSS_HASH`, `mbuf.rss`.

9.2.21 Inner RSS

Supports RX RSS hashing on Inner headers.

- **[uses] rte_flow_action_rss**: `level`.
- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_RSS_HASH`.
- **[provides] mbuf**: `mbuf.ol_flags:PKT_RX_RSS_HASH`, `mbuf.rss`.

9.2.22 RSS key update

Supports configuration of Receive Side Scaling (RSS) hash computation. Updating Receive Side Scaling (RSS) hash key.

- **[implements] eth_dev_ops**: `rss_hash_update`, `rss_hash_conf_get`.
- **[provides] rte_eth_dev_info**: `hash_key_size`.
- **[related] API**: `rte_eth_dev_rss_hash_update()`, `rte_eth_dev_rss_hash_conf_get()`.

9.2.23 RSS reta update

Supports updating Redirection Table of the Receive Side Scaling (RSS).

- **[implements] eth_dev_ops**: `reta_update`, `reta_query`.
- **[provides] rte_eth_dev_info**: `reta_size`.
- **[related] API**: `rte_eth_dev_rss_reta_update()`, `rte_eth_dev_rss_reta_query()`.

9.2.24 VMDq

Supports Virtual Machine Device Queues (VMDq).

- **[uses] user config**: `dev_conf.rxmode.mq_mode` = `ETH_MQ_RX_VMDQ_FLAG`.
- **[uses] user config**: `dev_conf.rx_adv_conf.vmdq_dcb_conf`.
- **[uses] user config**: `dev_conf.rx_adv_conf.vmdq_rx_conf`.
- **[uses] user config**: `dev_conf.tx_adv_conf.vmdq_dcb_tx_conf`.
- **[uses] user config**: `dev_conf.tx_adv_conf.vmdq_tx_conf`.

9.2.25 SR-IOV

Driver supports creating Virtual Functions.

- **[implements] rte_eth_dev_data**: `sriov`.

9.2.26 DCB

Supports Data Center Bridging (DCB).

- **[uses] user config**: `dev_conf.rxmode.mq_mode = ETH_MQ_RX_DCB_FLAG`.
- **[uses] user config**: `dev_conf.rx_adv_conf.vmdq_dcb_conf`.
- **[uses] user config**: `dev_conf.rx_adv_conf.dcb_rx_conf`.
- **[uses] user config**: `dev_conf.tx_adv_conf.vmdq_dcb_tx_conf`.
- **[uses] user config**: `dev_conf.tx_adv_conf.vmdq_tx_conf`.
- **[implements] eth_dev_ops**: `get_dcb_info`.
- **[related] API**: `rte_eth_dev_get_dcb_info()`.

9.2.27 VLAN filter

Supports filtering of a VLAN Tag identifier.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_VLAN_FILTER`.
- **[implements] eth_dev_ops**: `vlan_filter_set`.
- **[related] API**: `rte_eth_dev_vlan_filter()`.

9.2.28 Flow control

Supports configuring link flow control.

- **[implements] eth_dev_ops**: `flow_ctrl_get`, `flow_ctrl_set`, `priority_flow_ctrl_set`.
- **[related] API**: `rte_eth_dev_flow_ctrl_get()`, `rte_eth_dev_flow_ctrl_set()`, `rte_eth_dev_priority_flow_ctrl_set()`.

9.2.29 Flow API

Supports the DPDK Flow API for generic filtering.

- **[implements] eth_dev_ops**: `filter_ctrl:RTE_ETH_FILTER_GENERIC`.
- **[implements] rte_flow_ops**: `All`.

9.2.30 Rate limitation

Supports Tx rate limitation for a queue.

- **[implements] eth_dev_ops**: `set_queue_rate_limit`.
- **[related] API**: `rte_eth_set_queue_rate_limit()`.

9.2.31 Traffic mirroring

Supports adding traffic mirroring rules.

- **[implements] eth_dev_ops**: `mirror_rule_set`, `mirror_rule_reset`.
- **[related] API**: `rte_eth_mirror_rule_set()`, `rte_eth_mirror_rule_reset()`.

9.2.32 Inline crypto

Supports inline crypto processing defined by rte_security library to perform crypto operations of security protocol while packet is received in NIC. NIC is not aware of protocol operations. See Security library and PMD documentation for more details.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_SECURITY`,
- **[uses] rte_eth_txconf,rte_eth_txmode**: `offloads:DEV_TX_OFFLOAD_SECURITY`.
- **[implements] rte_security_ops**: `session_create`, `session_update`, `session_stats_get`, `session_destroy`, `set_pkt_metadata`, `capabilities_get`.
- **[provides] rte_eth_dev_info**: `rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_SECURITY`, `tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_SECURITY`.
- **[provides] mbuf**: `mbuf.ol_flags:PKT_RX_SEC_OFFLOAD`, `mbuf.ol_flags:PKT_TX_SEC_OFFLOAD`, `mbuf.ol_flags:PKT_RX_SEC_OFFLOAD_FAILED`.
- **[provides] rte_security_ops, capabilities_get**: `action: RTE_SECURITY_ACTION_TYPE_INLINE_CRYPTO`

9.2.33 Inline protocol

Supports inline protocol processing defined by rte_security library to perform protocol processing for the security protocol (e.g. IPsec, MACSEC) while the packet is received at NIC. The NIC is capable of understanding the security protocol operations. See security library and PMD documentation for more details.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_SECURITY`,
- **[uses] rte_eth_txconf,rte_eth_txmode**: `offloads:DEV_TX_OFFLOAD_SECURITY`.
- **[implements] rte_security_ops**: `session_create`, `session_update`, `session_stats_get`, `session_destroy`, `set_pkt_metadata`, `get_userdata`, `capabilities_get`.
- **[provides] rte_eth_dev_info**: `rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_SECURITY`, `tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_SECURITY`.
- **[provides] mbuf**: `mbuf.ol_flags:PKT_RX_SEC_OFFLOAD`, `mbuf.ol_flags:PKT_TX_SEC_OFFLOAD`, `mbuf.ol_flags:PKT_RX_SEC_OFFLOAD_FAILED`.
- **[provides] rte_security_ops, capabilities_get**: `action: RTE_SECURITY_ACTION_TYPE_INLINE_PROTOCOL`

9.2.34 CRC offload

Supports CRC stripping by hardware. A PMD assumed to support CRC stripping by default. PMD should advertise if it supports keeping CRC.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_KEEP_CRC`.

9.2.35 VLAN offload

Supports VLAN offload to hardware.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_VLAN_STRIP`, `DEV_RX_OFFLOAD_VLAN_FILTER`,`DEV_RX_OFFLOAD_VLAN_EXTEND`.
- **[uses] rte_eth_txconf,rte_eth_txmode**: `offloads:DEV_TX_OFFLOAD_VLAN_INSERT`.
- **[uses] mbuf**: `mbuf.ol_flags:PKT_TX_VLAN`, `mbuf.vlan_tci`.
- **[implements] eth_dev_ops**: `vlan_offload_set`.
- **[provides] mbuf**: `mbuf.ol_flags:PKT_RX_VLAN_STRIPPED`, `mbuf.ol_flags:PKT_RX_VLAN` `mbuf.vlan_tci`.
- **[provides] rte_eth_dev_info**: `rx_offload_capa`,`rx_queue_offload_capa:DEV_RX_OFFLOAD_VLAN_STRIP`, `tx_offload_capa`,`tx_queue_offload_capa:DEV_TX_OFFLOAD_VLAN_INSERT`.
- **[related] API**: `rte_eth_dev_set_vlan_offload()`, `rte_eth_dev_get_vlan_offload()`.

9.2.36 QinQ offload

Supports QinQ (queue in queue) offload.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_QINQ_STRIP`.
- **[uses] rte_eth_txconf,rte_eth_txmode**: `offloads:DEV_TX_OFFLOAD_QINQ_INSERT`.
- **[uses] mbuf**: `mbuf.ol_flags:PKT_TX_QINQ`, `mbuf.vlan_tci_outer`.
- **[provides] mbuf**: `mbuf.ol_flags:PKT_RX_QINQ_STRIPPED`, `mbuf.ol_flags:PKT_RX_QINQ`, `mbuf.ol_flags:PKT_RX_VLAN_STRIPPED`, `mbuf.ol_flags:PKT_RX_VLAN` `mbuf.vlan_tci`, `mbuf.vlan_tci_outer`.
- **[provides] rte_eth_dev_info**: `rx_offload_capa`,`rx_queue_offload_capa:DEV_RX_OFFLOAD_QINQ_STRIP`, `tx_offload_capa`,`tx_queue_offload_capa:DEV_TX_OFFLOAD_QINQ_INSERT`.

9.2.37 L3 checksum offload

Supports L3 checksum offload.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_IPV4_CKSUM`.
- **[uses] rte_eth_txconf,rte_eth_txmode**: `offloads:DEV_TX_OFFLOAD_IPV4_CKSUM`.
- **[uses] mbuf**: `mbuf.ol_flags:PKT_TX_IP_CKSUM`, `mbuf.ol_flags:PKT_TX_IPV4` | `PKT_TX_IPV6`.
- **[uses] mbuf**: `mbuf.l2_len`, `mbuf.l3_len`.

- **[provides] mbuf**: mbuf.ol_flags:PKT_RX_IP_CKSUM_UNKNOWN | PKT_RX_IP_CKSUM_BAD | PKT_RX_IP_CKSUM_GOOD | PKT_RX_IP_CKSUM_NONE.
- **[provides] rte_eth_dev_info**: rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_IPV4_CKSUM, tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_IPV4_CKSUM.

9.2.38 L4 checksum offload

Supports L4 checksum offload.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: offloads:DEV_RX_OFFLOAD_UDP_CKSUM, DEV_RX_OFFLOAD_TCP_CKSUM,DEV_RX_OFFLOAD_SCTP_CKSUM.
- **[uses] rte_eth_txconf,rte_eth_txmode**: offloads:DEV_TX_OFFLOAD_UDP_CKSUM, DEV_TX_OFFLOAD_TCP_CKSUM,DEV_TX_OFFLOAD_SCTP_CKSUM.
- **[uses] mbuf**: mbuf.ol_flags:PKT_TX_IPV4 | PKT_TX_IPV6, mbuf.ol_flags:PKT_TX_L4_NO_CKSUM | PKT_TX_TCP_CKSUM | PKT_TX_SCTP_CKSUM | PKT_TX_UDP_CKSUM.
- **[uses] mbuf**: mbuf.l2_len, mbuf.l3_len.
- **[provides] mbuf**: mbuf.ol_flags:PKT_RX_L4_CKSUM_UNKNOWN | PKT_RX_L4_CKSUM_BAD | PKT_RX_L4_CKSUM_GOOD | PKT_RX_L4_CKSUM_NONE.
- **[provides] rte_eth_dev_info**: rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_UDP_CKSUM, DEV_RX_OFFLOAD_TCP_CKSUM,DEV_RX_OFFLOAD_SCTP_CKSUM, tx_offload_capa, tx_queue_offload_capa:DEV_TX_OFFLOAD_UDP_CKSUM,DEV_TX_OFFLOAD_TCP_CKSUM, DEV_TX_OFFLOAD_SCTP_CKSUM.

9.2.39 Timestamp offload

Supports Timestamp.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: offloads:DEV_RX_OFFLOAD_TIMESTAMP.
- **[provides] mbuf**: mbuf.ol_flags:PKT_RX_TIMESTAMP.
- **[provides] mbuf**: mbuf.timestamp.
- **[provides] rte_eth_dev_info**: rx_offload_capa,rx_queue_offload_capa: DEV_RX_OFFLOAD_TIMESTAMP.
- **[related] eth_dev_ops**: read_clock.

9.2.40 MACsec offload

Supports MACsec.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: offloads:DEV_RX_OFFLOAD_MACSEC_STRIP.
- **[uses] rte_eth_txconf,rte_eth_txmode**: offloads:DEV_TX_OFFLOAD_MACSEC_INSERT.
- **[uses] mbuf**: mbuf.ol_flags:PKT_TX_MACSEC.
- **[provides] rte_eth_dev_info**: rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_MACSEC_STRI tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_MACSEC_INSERT.

9.2.41 Inner L3 checksum

Supports inner packet L3 checksum.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_OUTER_IPV4_CKSUM`.
- **[uses] rte_eth_txconf,rte_eth_txmode**: `offloads:DEV_TX_OFFLOAD_OUTER_IPV4_CKSUM`.
- **[uses] mbuf**: `mbuf.ol_flags:PKT_TX_IP_CKSUM`, `mbuf.ol_flags:PKT_TX_IPV4 | PKT_TX_IPV6`, `mbuf.ol_flags:PKT_TX_OUTER_IP_CKSUM`, `mbuf.ol_flags:PKT_TX_OUTER_IPV4 | PKT_TX_OUTER_IPV6`.
- **[uses] mbuf**: `mbuf.outer_l2_len`, `mbuf.outer_l3_len`.
- **[provides] mbuf**: `mbuf.ol_flags:PKT_RX_EIP_CKSUM_BAD`.
- **[provides] rte_eth_dev_info**: `rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_OUTER_IPV4_ tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_OUTER_IPV4_CKSUM`.

9.2.42 Inner L4 checksum

Supports inner packet L4 checksum.

- **[uses] rte_eth_rxconf,rte_eth_rxmode**: `offloads:DEV_RX_OFFLOAD_OUTER_UDP_CKSUM`.
- **[provides] mbuf**: `mbuf.ol_flags:PKT_RX_OUTER_L4_CKSUM_UNKNOWN | PKT_RX_OUTER_L4_CKSUM_BAD | PKT_RX_OUTER_L4_CKSUM_GOOD | PKT_RX_OUTER_L4_CKSUM_INVALID`.
- **[uses] rte_eth_txconf,rte_eth_txmode**: `offloads:DEV_TX_OFFLOAD_OUTER_UDP_CKSUM`.
- **[uses] mbuf**: `mbuf.ol_flags:PKT_TX_OUTER_IPV4 | PKT_TX_OUTER_IPV6`. `mbuf.ol_flags:PKT_TX_OUTER_UDP_CKSUM`.
- **[uses] mbuf**: `mbuf.outer_l2_len`, `mbuf.outer_l3_len`.
- **[provides] rte_eth_dev_info**: `rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_OUTER_UDP_C tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_OUTER_UDP_CKSUM`.

9.2.43 Packet type parsing

Supports packet type parsing and returns a list of supported types. Allows application to set ptypes it is interested in.

- **[implements] eth_dev_ops**: `dev_supported_ptypes_get`,
- **[related] API**: `rte_eth_dev_get_supported_ptypes()`, `rte_eth_dev_set_ptypes()`, `dev_ptypes_set`.
- **[provides] mbuf**: `mbuf.packet_type`.

9.2.44 Timesync

Supports IEEE1588/802.1AS timestamping.

- **[implements] eth_dev_ops**: `timesync_enable`, `timesync_disable` `timesync_read_rx_timestamp`, `timesync_read_tx_timestamp`, `timesync_adjust_time`, `timesync_read_time`, `timesync_write_time`.
- **[related] API**: `rte_eth_timesync_enable()`, `rte_eth_timesync_disable()`, `rte_eth_timesync_read_rx_timestamp()`, `rte_eth_timesync_read_tx_timestamp`, `rte_eth_timesync_adjust_time()`, `rte_eth_timesync_read_time()`, `rte_eth_timesync_write_time()`.

9.2.45 Rx descriptor status

Supports check the status of a Rx descriptor. When `rx_descriptor_status` is used, status can be "Available", "Done" or "Unavailable". When `rx_descriptor_done` is used, status can be "DD bit is set" or "DD bit is not set".

- **[implements] eth_dev_ops**: `rx_descriptor_status`.
- **[related] API**: `rte_eth_rx_descriptor_status()`.
- **[implements] eth_dev_ops**: `rx_descriptor_done`.
- **[related] API**: `rte_eth_rx_descriptor_done()`.

9.2.46 Tx descriptor status

Supports checking the status of a Tx descriptor. Status can be "Full", "Done" or "Unavailable."

- **[implements] eth_dev_ops**: `tx_descriptor_status`.
- **[related] API**: `rte_eth_tx_descriptor_status()`.

9.2.47 Basic stats

Support basic statistics such as: ipackets, opackets, ibytes, obytes, imissed, ierrors, oerrors, rx_nombuf.

And per queue stats: q_ipackets, q_opackets, q_ibytes, q_obytes, q_errors.

These apply to all drivers.

- **[implements] eth_dev_ops**: `stats_get`, `stats_reset`.
- **[related] API**: `rte_eth_stats_get`, `rte_eth_stats_reset()`.

9.2.48 Extended stats

Supports Extended Statistics, changes from driver to driver.

- **[implements] eth_dev_ops**: `xstats_get`, `xstats_reset`, `xstats_get_names`.
- **[implements] eth_dev_ops**: `xstats_get_by_id`, `xstats_get_names_by_id`.
- **[related] API**: `rte_eth_xstats_get()`, `rte_eth_xstats_reset()`, `rte_eth_xstats_get_names`, `rte_eth_xstats_get_by_id()`, `rte_eth_xstats_get_names_by_id()`, `rte_eth_xstats_get_id_by_name()`.

9.2.49 Stats per queue

Supports configuring per-queue stat counter mapping.

- **[implements] eth_dev_ops**: `queue_stats_mapping_set`.
- **[related] API**: `rte_eth_dev_set_rx_queue_stats_mapping()`, `rte_eth_dev_set_tx_queue_stats_mapping()`.

9.2.50 FW version

Supports getting device hardware firmware information.

- **[implements] eth_dev_ops**: `fw_version_get`.
- **[related] API**: `rte_eth_dev_fw_version_get()`.

9.2.51 EEPROM dump

Supports getting/setting device eeprom data.

- **[implements] eth_dev_ops**: `get_eeprom_length`, `get_eeprom`, `set_eeprom`.
- **[related] API**: `rte_eth_dev_get_eeprom_length()`, `rte_eth_dev_get_eeprom()`, `rte_eth_dev_set_eeprom()`.

9.2.52 Module EEPROM dump

Supports getting information and data of plugin module eeprom.

- **[implements] eth_dev_ops**: `get_module_info`, `get_module_eeprom`.
- **[related] API**: `rte_eth_dev_get_module_info()`, `rte_eth_dev_get_module_eeprom()`.

9.2.53 Registers dump

Supports retrieving device registers and registering attributes (number of registers and register size).

- **[implements] eth_dev_ops**: `get_reg`.
- **[related] API**: `rte_eth_dev_get_reg_info()`.

9.2.54 LED

Supports turning on/off a software controllable LED on a device.

- **[implements] eth_dev_ops**: `dev_led_on`, `dev_led_off`.
- **[related] API**: `rte_eth_led_on()`, `rte_eth_led_off()`.

9.2.55 Multiprocess aware

Driver can be used for primary-secondary process model.

9.2.56 BSD nic_uio

BSD `nic_uio` module supported.

9.2.57 Linux UIO

Works with `igb_uio` kernel module.

- **[provides] RTE_PMD_REGISTER_KMOD_DEP**: `igb_uio`.

9.2.58 Linux VFIO

Works with `vfio-pci` kernel module.

- **[provides] RTE_PMD_REGISTER_KMOD_DEP**: `vfio-pci`.

9.2.59 Other kdrv

Kernel module other than above ones supported.

9.2.60 ARMv7

Support armv7 architecture.

Use `defconfig_arm-armv7a-*-*`.

9.2.61 ARMv8

Support armv8a (64bit) architecture.

Use `defconfig_arm64-armv8a-*-*`

9.2.62 Power8

Support PowerPC architecture.

Use `defconfig_ppc_64-power8-*-*`

9.2.63 x86-32

Support 32bits x86 architecture.

Use `defconfig_x86_x32-native-*-*` and `defconfig_i686-native-*-*`.

9.2.64 x86-64

Support 64bits x86 architecture.

Use `defconfig_x86_64-native-*-*`.

9.2.65 Usage doc

Documentation describes usage.

See `doc/guides/nics/*.rst`

9.2.66 Design doc

Documentation describes design.

See `doc/guides/nics/*.rst`.

9.2.67 Perf doc

Documentation describes performance values.

See `dpdk.org/doc/perf/*`.

9.2.68 Runtime Rx queue setup

Supports Rx queue setup after device started.

- **[provides] rte_eth_dev_info**: `dev_capa:RTE_ETH_DEV_CAPA_RUNTIME_RX_QUEUE_SETUP`.
- **[related] API**: `rte_eth_dev_info_get()`.

9.2.69 Runtime Tx queue setup

Supports Tx queue setup after device started.

- **[provides] rte_eth_dev_info**: `dev_capa:RTE_ETH_DEV_CAPA_RUNTIME_TX_QUEUE_SETUP`.
- **[related] API**: `rte_eth_dev_info_get()`.

9.2.70 Burst mode info

Supports to get Rx/Tx packet burst mode information.

- **[implements] eth_dev_ops**: `rx_burst_mode_get`, `tx_burst_mode_get`.
- **[related] API**: `rte_eth_rx_burst_mode_get()`, `rte_eth_tx_burst_mode_get()`.

9.2.71 Other dev ops not represented by a Feature

- `rxq_info_get`
- `txq_info_get`
- `vlan_tpid_set`
- `vlan_strip_queue_set`
- `vlan_pvid_set`
- `rx_queue_count`
- `l2_tunnel_offload_set`
- `uc_hash_table_set`
- `uc_all_hash_table_set`
- `udp_tunnel_port_add`
- `udp_tunnel_port_del`
- `l2_tunnel_eth_type_conf`
- `l2_tunnel_offload_set`
- `tx_pkt_prepare`

9.3 Compiling and testing a PMD for a NIC

This section demonstrates how to compile and run a Poll Mode Driver (PMD) for the available Network Interface Cards in DPDK using TestPMD.

TestPMD is one of the reference applications distributed with the DPDK. Its main purpose is to forward packets between Ethernet ports on a network interface and as such is the best way to test a PMD.

Refer to the *testpmd application user guide* for detailed information on how to build and run testpmd.

9.3.1 Driver Compilation

To compile a PMD for a platform, run make with appropriate target as shown below. Use "make" command in Linux and "gmake" in FreeBSD. This will also build testpmd.

To check available targets:

```
cd <DPDK-source-directory>
make showconfigs
```

Example output:

```
arm-armv7a-linux-gcc
arm64-armv8a-linux-gcc
arm64-dpaa-linux-gcc
arm64-thunderx-linux-gcc
arm64-xgene1-linux-gcc
i686-native-linux-gcc
i686-native-linux-icc
ppc_64-power8-linux-gcc
x86_64-native-freebsd-clang
x86_64-native-freebsd-gcc
x86_64-native-linux-clang
x86_64-native-linux-gcc
x86_64-native-linux-icc
x86_x32-native-linux-gcc
```

To compile a PMD for Linux x86_64 gcc target, run the following "make" command:

```
make install T=x86_64-native-linux-gcc
```

Use ARM (ThunderX, DPAA, X-Gene) or PowerPC target for respective platform.

For more information, refer to the *Getting Started Guide for Linux* or *Getting Started Guide for FreeBSD* depending on your platform.

9.3.2 Running testpmd in Linux

This section demonstrates how to setup and run `testpmd` in Linux.

1. Mount huge pages:

   ```
   mkdir /mnt/huge
   mount -t hugetlbfs nodev /mnt/huge
   ```

2. Request huge pages:

   Hugepage memory should be reserved as per application requirement. Check hugepage size configured in the system and calculate the number of pages required.

   To reserve 1024 pages of 2MB:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

   Note: Check `/proc/meminfo` to find system hugepage size:

```
grep "Hugepagesize:" /proc/meminfo
```

   Example output:

```
Hugepagesize:       2048 kB
```

3. Load `igb_uio` or `vfio-pci` driver:

```
modprobe uio
insmod ./x86_64-native-linux-gcc/kmod/igb_uio.ko
```

   or

```
modprobe vfio-pci
```

4. Setup VFIO permissions for regular users before binding to `vfio-pci`:

```
sudo chmod a+x /dev/vfio

sudo chmod 0666 /dev/vfio/*
```

5. Bind the adapters to `igb_uio` or `vfio-pci` loaded in the previous step:

```
./usertools/dpdk-devbind.py --bind igb_uio DEVICE1 DEVICE2 ...
```

   Or setup VFIO permissions for regular users and then bind to `vfio-pci`:

```
./usertools/dpdk-devbind.py --bind vfio-pci DEVICE1 DEVICE2 ...
```

   Note: DEVICE1, DEVICE2 are specified via PCI "domain:bus:slot.func" syntax or "bus:slot.func" syntax.

6. Start `testpmd` with basic parameters:

```
./x86_64-native-linux-gcc/app/testpmd -l 0-3 -n 4 -- -i
```

   Successful execution will show initialization messages from EAL, PMD and testpmd application. A prompt will be displayed at the end for user commands as interactive mode (`-i`) is on.

```
testpmd>
```

   Refer to the *testpmd runtime functions* for a list of available commands.

Note: When `testpmd` is built with shared library, use option `-d` to load the dynamic PMD for `rte_eal_init`.

9.4 AF_PACKET Poll Mode Driver

The AF_PACKET socket in Linux allows an application to receive and send raw packets. This Linux-specific PMD driver binds to an AF_PACKET socket and allows a DPDK application to send and receive raw packets through the Kernel.

In order to improve Rx and Tx performance this implementation makes use of PACKET_MMAP, which provides a mmap'ed ring buffer, shared between user space and kernel, that's used to send and receive packets. This helps reducing system calls and the copies needed between user space and Kernel.

The PACKET_FANOUT_HASH behavior of AF_PACKET is used for frame reception.

9.4.1 Options and inherent limitations

The following options can be provided to set up an af_packet port in DPDK. Some of these, in turn, will be used to configure the PACKET_MMAP settings.

- `iface` - name of the Kernel interface to attach to (required);
- `qpairs` - number of Rx and Tx queues (optional, default 1);
- `qdisc_bypass` - set PACKET_QDISC_BYPASS option in AF_PACKET (optional, disabled by default);
- `blocksz` - PACKET_MMAP block size (optional, default 4096);
- `framesz` - PACKET_MMAP frame size (optional, default 2048B; Note: multiple of 16B);
- `framecnt` - PACKET_MMAP frame count (optional, default 512).

Because this implementation is based on PACKET_MMAP, and PACKET_MMAP has its own pre-requisites, it should be noted that the inner workings of PACKET_MMAP should be carefully considered before modifying some of these options (namely, `blocksz`, `framesz` and `framecnt` above).

As an example, if one changes `framesz` to be 1024B, it is expected that `blocksz` is set to at least 1024B as well (although 2048B in this case would allow two "frames" per "block").

This restriction happens because PACKET_MMAP expects each single "frame" to fit inside of a "block". And although multiple "frames" can fit inside of a single "block", a "frame" may not span across two "blocks".

For the full details behind PACKET_MMAP's structures and settings, consider reading the PACKET_MMAP documentation in the Kernel.

9.4.2 Prerequisites

This is a Linux-specific PMD, thus the following prerequisites apply:

- A Linux Kernel;
- A Kernel bound interface to attach to (e.g. a tap interface).

9.4.3 Set up an af_packet interface

The following example will set up an af_packet interface in DPDK with the default options described above (blocksz=4096B, framesz=2048B and framecnt=512):

```
--vdev=eth_af_packet0,iface=tap0,blocksz=4096,framesz=2048,framecnt=512,qpairs=1,qdisc_bypass=0
```

9.5 AF_XDP Poll Mode Driver

AF_XDP is an address family that is optimized for high performance packet processing. AF_XDP sockets enable the possibility for XDP program to redirect packets to a memory buffer in userspace.

For the full details behind AF_XDP socket, you can refer to AF_XDP documentation in the Kernel.

This Linux-specific PMD driver creates the AF_XDP socket and binds it to a specific netdev queue, it allows a DPDK application to send and receive raw packets through the socket which would bypass the kernel network stack. Current implementation only supports single queue, multi-queues feature will be added later.

AF_XDP PMD enables need_wakeup flag by default if it is supported. This need_wakeup feature is used to support executing application and driver on the same core efficiently. This feature not only has a large positive performance impact for the one core case, but also does not degrade 2 core performance and actually improves it for Tx heavy workloads.

9.5.1 Options

The following options can be provided to set up an af_xdp port in DPDK.

- `iface` - name of the Kernel interface to attach to (required);
- `start_queue` - starting netdev queue id (optional, default 0);
- `queue_count` - total netdev queue number (optional, default 1);

9.5.2 Prerequisites

This is a Linux-specific PMD, thus the following prerequisites apply:

- A Linux Kernel (version > v4.18) with XDP sockets configuration enabled;
- libbpf (within kernel version > v5.1-rc4) with latest af_xdp support installed, User can install libbpf via *make install_lib && make install_headers* in <kernel src tree>/tools/lib/bpf;
- A Kernel bound interface to attach to;
- For need_wakeup feature, it requires kernel version later than v5.3-rc1;

- For PMD zero copy, it requires kernel version later than v5.4-rc1;

9.5.3 Set up an af_xdp interface

The following example will set up an af_xdp interface in DPDK:

```
--vdev net_af_xdp,iface=ens786f1
```

9.5.4 Limitations

- **MTU**

  The MTU of the AF_XDP PMD is limited due to the XDP requirement of one packet per page. In the PMD we report the maximum MTU for zero copy to be equal to the page size less the frame overhead introduced by AF_XDP (XDP HR = 256) and DPDK (frame headroom = 320). With a 4K page size this works out at 3520. However in practice this value may be even smaller, due to differences between the supported RX buffer sizes of the underlying kernel netdev driver.

  For example, the largest RX buffer size supported by the underlying kernel driver which is less than the page size (4096B) may be 3072B. In this case, the maximum MTU value will be at most 3072, but likely even smaller than this, once relevant headers are accounted for eg. Ethernet and VLAN.

  To determine the actual maximum MTU value of the interface you are using with the AF_XDP PMD, consult the documentation for the kernel driver.

  Note: The AF_XDP PMD will fail to initialise if an MTU which violates the driver's conditions as above is set prior to launching the application.

9.6 ARK Poll Mode Driver

The ARK PMD is a DPDK poll-mode driver for the Atomic Rules Arkville (ARK) family of devices.

More information can be found at the Atomic Rules website.

9.6.1 Overview

The Atomic Rules Arkville product is DPDK and AXI compliant product that marshals packets across a PCIe conduit between host DPDK mbufs and FPGA AXI streams.

The ARK PMD, and the spirit of the overall Arkville product, has been to take the DPDK API/ABI as a fixed specification; then implement much of the business logic in FPGA RTL circuits. The approach of *working backwards* from the DPDK API/ABI and having the GPP host software *dictate*, while the FPGA hardware *copes*, results in significant performance gains over a naive implementation.

While this document describes the ARK PMD software, it is helpful to understand what the FPGA hardware is and is not. The Arkville RTL component provides a single PCIe Physical Function (PF) supporting some number of RX/Ingress and TX/Egress Queues. The ARK PMD controls the Arkville core through a dedicated opaque Core BAR (CBAR). To allow users full freedom for their own FPGA application IP, an independent FPGA Application BAR (ABAR) is provided.

One popular way to imagine Arkville's FPGA hardware aspect is as the FPGA PCIe-facing side of a so-called Smart NIC. The Arkville core does not contain any MACs, and is link-speed independent, as well as agnostic to the number of physical ports the application chooses to use. The ARK driver exposes the familiar PMD interface to allow packet movement to and from mbufs across multiple queues.

However FPGA RTL applications could contain a universe of added functionality that an Arkville RTL core does not provide or can not anticipate. To allow for this expectation of user-defined innovation, the ARK PMD provides a dynamic mechanism of adding capabilities without having to modify the ARK PMD.

The ARK PMD is intended to support all instances of the Arkville RTL Core, regardless of configuration, FPGA vendor, or target board. While specific capabilities such as number of physical hardware queue-pairs are negotiated; the driver is designed to remain constant over a broad and extendable feature set.

Intentionally, Arkville by itself DOES NOT provide common NIC capabilities such as offload or receive-side scaling (RSS). These capabilities would be viewed as a gate-level "tax" on Green-box FPGA applications that do not require such function. Instead, they can be added as needed with essentially no overhead to the FPGA Application.

The ARK PMD also supports optional user extensions, through dynamic linking. The ARK PMD user extensions are a feature of Arkville's DPDK net/ark poll mode driver, allowing users to add their own code to extend the net/ark functionality without having to make source code changes to the driver. One motivation for this capability is that while DPDK provides a rich set of functions to interact with NIC-like capabilities (e.g. MAC addresses and statistics), the Arkville RTL IP does not include a MAC. Users can supply their own MAC or custom FPGA applications, which may require control from the PMD. The user extension is the means providing the control between the user's FPGA application and the existing DPDK features via the PMD.

9.6.2 Device Parameters

The ARK PMD supports device parameters that are used for packet routing and for internal packet generation and packet checking. This section describes the supported parameters. These features are primarily used for diagnostics, testing, and performance verification under the guidance of an Arkville specialist. The nominal use of Arkville does not require any configuration using these parameters.

"Pkt_dir"

The Packet Director controls connectivity between Arkville's internal hardware components. The features of the Pkt_dir are only used for diagnostics and testing; it is not intended for nominal use. The full set of features are not published at this level.

Format: Pkt_dir=0x00110F10

"Pkt_gen"

The packet generator parameter takes a file as its argument. The file contains configuration parameters used internally for regression testing and are not intended to be published at this level. The packet generator is an internal Arkville hardware component.

Format: Pkt_gen=./config/pg.conf

"Pkt_chkr"

The packet checker parameter takes a file as its argument. The file contains configuration parameters used internally for regression testing and are not intended to be published at this level. The packet checker is an internal Arkville hardware component.

Format: Pkt_chkr=./config/pc.conf

9.6.3 Data Path Interface

Ingress RX and Egress TX operation is by the nominal DPDK API . The driver supports single-port, multi-queue for both RX and TX.

9.6.4 Configuration Information

DPDK Configuration Parameters

The following configuration options are available for the ARK PMD:

- **CONFIG_RTE_LIBRTE_ARK_PMD** (default y): Enables or disables inclusion of the ARK PMD driver in the DPDK compilation.
- **CONFIG_RTE_LIBRTE_ARK_PAD_TX** (default y): When enabled TX packets are padded to 60 bytes to support downstream MACS.
- **CONFIG_RTE_LIBRTE_ARK_DEBUG_RX** (default n): Enables or disables debug logging and internal checking of RX ingress logic within the ARK PMD driver.
- **CONFIG_RTE_LIBRTE_ARK_DEBUG_TX** (default n): Enables or disables debug logging and internal checking of TX egress logic within the ARK PMD driver.
- **CONFIG_RTE_LIBRTE_ARK_DEBUG_STATS** (default n): Enables or disables debug logging of detailed packet and performance statistics gathered in the PMD and FPGA.
- **CONFIG_RTE_LIBRTE_ARK_DEBUG_TRACE** (default n): Enables or disables debug logging of detailed PMD events and status.

9.6.5 Building DPDK

See the *DPDK Getting Started Guide for Linux* for instructions on how to build DPDK.

By default the ARK PMD library will be built into the DPDK library.

For configuring and using UIO and VFIO frameworks, please also refer *the documentation that comes with DPDK suite*.

9.6.6 Supported ARK RTL PCIe Instances

ARK PMD supports the following Arkville RTL PCIe instances including:

- `1d6c:100d` - AR-ARKA-FX0 [Arkville 32B DPDK Data Mover]
- `1d6c:100e` - AR-ARKA-FX1 [Arkville 64B DPDK Data Mover]

9.6.7 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in `System Requirements` section of *the DPDK documentation* or refer to *DPDK Release Notes*. ARM and PowerPC architectures are not supported at this time.

9.6.8 Supported Features

- Dynamic ARK PMD extensions
- Multiple receive and transmit queues
- Jumbo frames up to 9K
- Hardware Statistics

9.6.9 Unsupported Features

Features that may be part of, or become part of, the Arkville RTL IP that are not currently supported or exposed by the ARK PMD include:

- PCIe SR-IOV Virtual Functions (VFs)
- Arkville's Packet Generator Control and Status
- Arkville's Packet Director Control and Status
- Arkville's Packet Checker Control and Status
- Arkville's Timebase Management

9.6.10 Pre-Requisites

1. Prepare the system as recommended by DPDK suite. This includes environment variables, hugepages configuration, tool-chains and configuration
2. Insert igb_uio kernel module using the command 'modprobe igb_uio'
3. Bind the intended ARK device to igb_uio module

At this point the system should be ready to run DPDK applications. Once the application runs to completion, the ARK PMD can be detached from igb_uio if necessary.

9.6.11 Usage Example

Follow instructions available in the document *compiling and testing a PMD for a NIC* to launch **testpmd** with Atomic Rules ARK devices managed by librte_pmd_ark.

Example output:

```
[...]
EAL: PCI device 0000:01:00.0 on NUMA socket -1
EAL:   probe driver: 1d6c:100e rte_ark_pmd
EAL:   PCI memory mapped at 0x7f9b6c400000
PMD: eth_ark_dev_init(): Initializing 0:2:0.1
```

(continues on next page)

(continued from previous page)

```
ARKP PMD CommitID: 378f3a67
Configuring Port 0 (socket 0)
Port 0: DC:3C:F6:00:00:01
Checking link statuses...
Port 0 Link Up - speed 100000 Mbps - full-duplex
Done
testpmd>
```

9.7 Aquantia Atlantic DPDK Driver

Atlantic DPDK driver provides DPDK support for Aquantia's AQtion family of chipsets: AQC107/AQC108/AQC109

More information can be found at Aquantia Official Website.

9.7.1 Supported features

- Base L2 features
- Promiscuous mode
- Multicast mode
- Port statistics
- RSS (Receive Side Scaling)
- Checksum offload
- Jumbo Frame up to 16K
- MACSEC offload

9.7.2 Experimental API features

- MACSEC PMD API is considered as experimental and is subject to change/removal in next DPDK releases.

9.7.3 Configuration Information

- `CONFIG_RTE_LIBRTE_ATLANTIC_PMD` (default `y`)

Application Programming Interface

Limitations or Known issues

Statistics

MTU setting

Atlantic NIC supports up to 16K jumbo frame size

Supported Chipsets and NICs

- Aquantia AQtion AQC107 10 Gigabit Ethernet Controller
- Aquantia AQtion AQC108 5 Gigabit Ethernet Controller
- Aquantia AQtion AQC109 2.5 Gigabit Ethernet Controller

9.8 AVP Poll Mode Driver

The Accelerated Virtual Port (AVP) device is a shared memory based device only available on virtualization platforms from Wind River Systems. The Wind River Systems virtualization platform currently uses QEMU/KVM as its hypervisor and as such provides support for all of the QEMU supported virtual and/or emulated devices (e.g., virtio, e1000, etc.). The platform offers the virtio device type as the default device when launching a virtual machine or creating a virtual machine port. The AVP device is a specialized device available to customers that require increased throughput and decreased latency to meet the demands of their performance focused applications.

The AVP driver binds to any AVP PCI devices that have been exported by the Wind River Systems QEMU/KVM hypervisor. As a user of the DPDK driver API it supports a subset of the full Ethernet device API to enable the application to use the standard device configuration functions and packet receive/transmit functions.

These devices enable optimized packet throughput by bypassing QEMU and delivering packets directly to the virtual switch via a shared memory mechanism. This provides DPDK applications running in virtual machines with significantly improved throughput and latency over other device types.

The AVP device implementation is integrated with the QEMU/KVM live-migration mechanism to allow applications to seamlessly migrate from one hypervisor node to another with minimal packet loss.

9.8.1 Features and Limitations of the AVP PMD

The AVP PMD driver provides the following functionality.

- Receive and transmit of both simple and chained mbuf packets,
- Chained mbufs may include up to 5 chained segments,
- Up to 8 receive and transmit queues per device,
- Only a single MAC address is supported,
- The MAC address cannot be modified,

- The maximum receive packet length is 9238 bytes,
- VLAN header stripping and inserting,
- Promiscuous mode
- VM live-migration
- PCI hotplug insertion and removal

9.8.2 Prerequisites

The following prerequisites apply:

- A virtual machine running in a Wind River Systems virtualization environment and configured with at least one neutron port defined with a vif-model set to "avp".

9.8.3 Launching a VM with an AVP type network attachment

The following example will launch a VM with three network attachments. The first attachment will have a default vif-model of "virtio". The next two network attachments will have a vif-model of "avp" and may be used with a DPDK application which is built to include the AVP PMD driver.

```
nova boot --flavor small --image my-image \
   --nic net-id=${NETWORK1_UUID} \
   --nic net-id=${NETWORK2_UUID},vif-model=avp \
   --nic net-id=${NETWORK3_UUID},vif-model=avp \
   --security-group default my-instance1
```

9.9 AXGBE Poll Mode Driver

The AXGBE poll mode driver library (**librte_pmd_axgbe**) implements support for AMD 10 Gbps family of adapters. It is compiled and tested in standard linux distro like Ubuntu.

Detailed information about SoCs that use these devices can be found here:

- AMD EPYC™ EMBEDDED 3000 family.

9.9.1 Supported Features

AXGBE PMD has support for:

- Base L2 features
- TSS (Transmit Side Scaling)
- Promiscuous mode
- Port statistics
- Multicast mode
- RSS (Receive Side Scaling)
- Checksum offload

- Jumbo Frame up to 9K

9.9.2 Configuration Information

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_AXGBE_PMD` (default **y**)

  Toggle compilation of axgbe PMD.

- `CONFIG_RTE_LIBRTE_AXGBE_PMD_DEBUG` (default **n**)

  Toggle display for PMD debug related messages.

9.9.3 Building DPDK

See the *DPDK Getting Started Guide for Linux* for instructions on how to build DPDK.

By default the AXGBE PMD library will be built into the DPDK library.

For configuring and using UIO frameworks, please also refer *the documentation that comes with DPDK suite*.

9.9.4 Prerequisites and Pre-conditions

- Prepare the system as recommended by DPDK suite.
- Bind the intended AMD device to `igb_uio` or `vfio-pci` module.

Now system is ready to run DPDK application.

9.9.5 Usage Example

Refer to the document *compiling and testing a PMD for a NIC* for details.

Example output:

```
[...]
EAL: PCI device 0000:02:00.4 on NUMA socket 0
EAL:   probe driver: 1022:1458 net_axgbe
Interactive-mode selected
USER1: create a new mbuf pool <mbuf_pool_socket_0>: n=171456, size=2176, socket=0
USER1: create a new mbuf pool <mbuf_pool_socket_1>: n=171456, size=2176, socket=1
USER1: create a new mbuf pool <mbuf_pool_socket_2>: n=171456, size=2176, socket=2
USER1: create a new mbuf pool <mbuf_pool_socket_3>: n=171456, size=2176, socket=3
Configuring Port 0 (socket 0)
Port 0: 00:00:1A:1C:6A:17
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

9.10 BNX2X Poll Mode Driver

The BNX2X poll mode driver library (**librte_pmd_bnx2x**) implements support for **QLogic 578xx** 10/20 Gbps family of adapters as well as their virtual functions (VF) in SR-IOV context. It is supported on several standard Linux distros like RHEL and SLES. It is compile-tested under FreeBSD OS.

More information can be found at QLogic Corporation's Official Website.

9.10.1 Supported Features

BNX2X PMD has support for:

- Base L2 features
- Unicast/multicast filtering
- Promiscuous mode
- Port hardware statistics
- SR-IOV VF

9.10.2 Non-supported Features

The features not yet supported include:

- TSS (Transmit Side Scaling)
- RSS (Receive Side Scaling)
- LRO/TSO offload
- Checksum offload
- SR-IOV PF
- Rx TX scatter gather

9.10.3 Co-existence considerations

- QLogic 578xx CNAs support Ethernet, iSCSI and FCoE functionalities. These functionalities are supported using QLogic Linux kernel drivers bnx2x, cnic, bnx2i and bnx2fc. DPDK is supported on these adapters using bnx2x PMD.
- When SR-IOV is not enabled on the adapter, QLogic Linux kernel drivers (bnx2x, cnic, bnx2i and bnx2fc) and bnx2x PMD can't be attached to different PFs on a given QLogic 578xx adapter. A given adapter needs to be completely used by DPDK or Linux drivers. Before binding DPDK driver to one or more PFs on the adapter, please make sure to unbind Linux drivers from all PFs of the adapter. If there are multiple adapters on the system, one or more adapters can be used by DPDK driver completely and other adapters can be used by Linux drivers completely.
- When SR-IOV is enabled on the adapter, Linux kernel drivers (bnx2x, cnic, bnx2i and bnx2fc) can be bound to the PFs of a given adapter and either bnx2x PMD or Linux drivers bnx2x can be bound to the VFs of the adapter.

9.10.4 Supported QLogic NICs

- 578xx

9.10.5 Prerequisites

- Requires firmware version **7.13.11.0**. It is included in most of the standard Linux distros. If it is not available visit linux-firmware git repository to get the required firmware.

9.10.6 Pre-Installation Configuration

Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_BNX2X_PMD` (default **n**)

  Toggle compilation of bnx2x driver. To use bnx2x PMD set this config parameter to 'y'. Also, in order for firmware binary to load user will need zlib devel package installed.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_TX` (default **n**)

  Toggle display of transmit fast path run-time messages.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_RX` (default **n**)

  Toggle display of receive fast path run-time messages.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_PERIODIC` (default **n**)

  Toggle display of register reads and writes.

9.10.7 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

9.10.8 Jumbo: Limitation

Rx descriptor limit for number of segments per MTU is set to 1. PMD doesn't support Jumbo Rx scatter gather. Some applciations can adjust mbuf_size based on this param and max_pkt_len.

For others, PMD detects the condition where Rx packet length cannot be held by configured mbuf size and logs the message.

Example output:

```
[...]
[bnx2x_recv_pkts:397(04:00.0:dpdk-port-0)] mbuf size 2048 is not enough to hold Rx␣
↪packet length more than 2046
```

9.10.9 SR-IOV: Prerequisites and sample Application Notes

This section provides instructions to configure SR-IOV with Linux OS.

1. Verify SR-IOV and ARI capabilities are enabled on the adapter using `lspci`:

```
lspci -s <slot> -vvv
```

   Example output:

```
[...]
Capabilities: [1b8 v1] Alternative Routing-ID Interpretation (ARI)
[...]
Capabilities: [1c0 v1] Single Root I/O Virtualization (SR-IOV)
[...]
Kernel driver in use: igb_uio
```

2. Load the kernel module:

```
modprobe bnx2x
```

   Example output:

```
systemd-udevd[4848]: renamed network interface eth0 to ens5f0
systemd-udevd[4848]: renamed network interface eth1 to ens5f1
```

3. Bring up the PF ports:

```
ifconfig ens5f0 up
ifconfig ens5f1 up
```

4. Create VF device(s):

   Echo the number of VFs to be created into "sriov_numvfs" sysfs entry of the parent PF.

   Example output:

```
echo 2 > /sys/devices/pci0000:00/0000:00:03.0/0000:81:00.0/sriov_numvfs
```

5. Assign VF MAC address:

   Assign MAC address to the VF using iproute2 utility. The syntax is: ip link set <PF iface> vf <VF id> mac <macaddr>

   Example output:

```
ip link set ens5f0 vf 0 mac 52:54:00:2f:9d:e8
```

6. PCI Passthrough:

   The VF devices may be passed through to the guest VM using virt-manager or virsh etc. bnx2x PMD should be used to bind the VF devices in the guest VM using the instructions outlined in the Application notes below.

7. Running testpmd: (Supply `--log-level="pmd.net.bnx2x.driver",7` to view informational messages):

   Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

Example output:

```
[...]
EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL:   probe driver: 14e4:168e rte_bnx2x_pmd
EAL:   PCI memory mapped at 0x7f14f6fe5000
EAL:   PCI memory mapped at 0x7f14f67e5000
EAL:   PCI memory mapped at 0x7f15fbd9b000
EAL: PCI device 0000:84:00.1 on NUMA socket 1
EAL:   probe driver: 14e4:168e rte_bnx2x_pmd
EAL:   PCI memory mapped at 0x7f14f5fe5000
EAL:   PCI memory mapped at 0x7f14f57e5000
EAL:   PCI memory mapped at 0x7f15fbd4f000
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: bnx2x_dev_tx_queue_setup(): fp[00] req_bd=512, thresh=512,
             usable_bd=1020, total_bd=1024,
                          tx_pages=4
PMD: bnx2x_dev_rx_queue_setup(): fp[00] req_bd=128, thresh=0,
             usable_bd=510, total_bd=512,
                          rx_pages=1, cq_pages=8
PMD: bnx2x_print_adapter_info():
[...]
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

9.11 BNXT Poll Mode Driver

The Broadcom BNXT PMD (**librte_pmd_bnxt**) implements support for adapters based on Ethernet controllers and SoCs belonging to the Broadcom BCM574XX/BCM575XX NetXtreme-E® Family of Ethernet Network Controllers, the Broadcom BCM588XX Stingray Family of Smart NIC Adapters, and the Broadcom StrataGX® BCM5873X Series of Communications Processors.

A complete list with links to reference material is in the Appendix section.

9.11.1 CPU Support

BNXT PMD supports multiple CPU architectures, including x86-32, x86-64, and ARMv8.

9.11.2 Kernel Dependency

BNXT PMD requires a kernel module (VFIO or UIO) for setting up a device, mapping device memory to userspace, registering interrupts, etc. VFIO is more secure than UIO, relying on IOMMU protection. UIO requires the IOMMU disabled or configured to pass-through mode.

Operating Systems supported:

- Red Hat Enterprise Linux release 8.1 (Ootpa)
- Red Hat Enterprise Linux release 8.0 (Ootpa)
- Red Hat Enterprise Linux Server release 7.7 (Maipo)

- Red Hat Enterprise Linux Server release 7.6 (Maipo)
- Red Hat Enterprise Linux Server release 7.5 (Maipo)
- Red Hat Enterprise Linux Server release 7.4 (Maipo)
- Red Hat Enterprise Linux Server release 7.3 (Maipo)
- Red Hat Enterprise Linux Server release 7.2 (Maipo)
- CentOS Linux release 8.0
- CentOS Linux release 7.7
- CentOS Linux release 7.6.1810
- CentOS Linux release 7.5.1804
- CentOS Linux release 7.4.1708
- Fedora 31
- FreeBSD 12.1
- Suse 15SP1
- Ubuntu 19.04
- Ubuntu 18.04
- Ubuntu 16.10
- Ubuntu 16.04
- Ubuntu 14.04

The BNXT PMD supports operating with:

- Linux vfio-pci
- Linux uio_pci_generic
- Linux igb_uio
- BSD nic_uio

9.11.3 Compiling BNXT PMD

To compile the BNXT PMD:

```
make config T=x86_64-native-linux-gcc && make // for x86-64
make config T=x86_32-native-linux-gcc && make // for x86-32
make config T=armv8a-linux-gcc && make // for ARMv8
```

Bind the device to one of the kernel modules listed above

```
./dpdk-devbind.py -b vfio-pci|igb_uio|uio_pci_generic bus_id:device_id.function_id
```

Load an application (e.g. testpmd) with a default configuration (e.g. a single TX /RX queue):

```
./testpmd -c 0xF -n 4 -- -i --portmask=0x1 --nb-cores=2
```

9.11.4 Running BNXT PMD

The BNXT PMD can run on PF or VF.

PCI-SIG Single Root I/O Virtualization (SR-IOV) involves the direct assignment of part of the network port resources to guest operating systems using the SR-IOV standard. NIC is logically distributed among multiple virtual machines (VMs), while still having global data in common to share with the PF and other VFs.

Sysadmin can create and configure VFs:

```
echo num_vfs > /sys/bus/pci/devices/domain_id:bus_id:device_id:function_id/sriov_numvfs
(ex) echo 4 > /sys/bus/pci/devices/0000:82:00:0/sriov_numvfs
```

Sysadmin also can change the VF property such as MAC address, transparent VLAN, TX rate limit, and trusted VF:

```
ip link set pf_id vf vf_id mac (mac_address) vlan (vlan_id) txrate (rate_value) trust␣
↪(enable|disable)
(ex) ip link set 0 vf 0 mac 00:11:22:33:44:55 vlan 0x100 txrate 100 trust disable
```

Running on VF

Flow Bifurcation

The Flow Bifurcation splits the incoming data traffic to user space applications (such as DPDK applications) and/or kernel space programs (such as the Linux kernel stack). It can direct some traffic, for example data plane traffic, to DPDK. Rest of the traffic, for example control plane traffic, would be redirected to the traditional Linux networking stack.

Refer to https://doc.dpdk.org/guides/howto/flow_bifurcation.html

Benefits of the flow bifurcation include:

- Better performance with less CPU overhead, as user application can directly access the NIC for data path
- NIC is still being controlled by the kernel, as control traffic is forwarded only to the kernel driver
- Control commands, e.g. ethtool, will work as usual

Running on a VF, the BXNT PMD supports the flow bifurcation with a combination of SR-IOV and packet classification and/or forwarding capability. In the simplest case of flow bifurcation, a PF driver configures a NIC to forward all user traffic directly to VFs with matching destination MAC address, while the rest of the traffic is forwarded to a PF. Note that the broadcast packets will be forwarded to both PF and VF.

```
(ex) ethtool --config-ntuple ens2f0 flow-type ether dst 00:01:02:03:00:01 vlan 10 vlan-mask␣
↪0xf000 action 0x100000000
```

Trusted VF

By default, VFs are *not* allowed to perform privileged operations, such as modifying the VF's MAC address in the guest. These security measures are designed to prevent possible attacks. However, when a DPDK application can be trusted (e.g., OVS-DPDK, here), these operations performed by a VF would be legitimate and can be allowed.

To enable VF to request "trusted mode," a new trusted VF concept was introduced in Linux kernel 4.4 and allowed VFs to become "trusted" and perform some privileged operations.

The BNXT PMD supports the trusted VF mode of operation. Only a PF can enable the trusted attribute on the VF. It is preferable to enable the Trusted setting on a VF before starting applications. However, the BNXT PMD handles dynamic changes in trusted settings as well.

Note that control commands, e.g., ethtool, will work via the kernel PF driver, *not* via the trusted VF driver.

Operations supported by trusted VF:

- MAC address configuration
- Flow rule creation

Operations *not* supported by trusted VF:

- Firmware upgrade
- Promiscuous mode setting

Running on PF

Unlike the VF when BNXT PMD runs on a PF there are no restrictions placed on the features which the PF can enable or request. In a multiport NIC, each port will have a corresponding PF. Also depending on the configuration of the NIC there can be more than one PF associated per port. A sysadmin can load the kernel driver on one PF, and run BNXT PMD on the other PF or run the PMD on both the PFs. In such cases, the firmware picks one of the PFs as a master PF.

Much like in the trusted VF, the DPDK application must be *trusted* and expected to be *well-behaved*.

9.11.5 Features

The BNXT PMD supports the following features:

- **Port Control**
  - Port MTU
  - LED
  - Flow Control and Autoneg
- **Packet Filtering**
  - Unicast MAC Filter
  - Multicast MAC Filter
  - VLAN Filtering

  - Allmulticast Mode
  - Promiscuous Mode
- **Stateless Offloads**
  - CRC Offload
  - Checksum Offload (IPv4, TCP, and UDP)
  - Multi-Queue (TSS and RSS)
  - Segmentation and Reassembly (TSO and LRO)
- VLAN insert strip
- Stats Collection
- Generic Flow Offload

Port Control

Port MTU: BNXT PMD supports the MTU (Maximum Transmission Unit) up to 9,574 bytes:

```
testpmd> port config mtu (port_id) mtu_value
testpmd> show port info (port_id)
```

LED: Application tunes on (or off) a port LED, typically for a port identification:

```
int rte_eth_led_on (uint16_t port_id)
int rte_eth_led_off (uint16_t port_id)
```

Flow Control and Autoneg: Application tunes on (or off) flow control and/or auto-negotiation on a port:

```
testpmd> set flow_ctrl rx (on|off) (port_id)
testpmd> set flow_ctrl tx (on|off) (port_id)
testpmd> set flow_ctrl autoneg (on|off) (port_id)
```

Note that the BNXT PMD does *not* support some options and ignores them when requested:

- high_water
- low_water
- pause_time
- mac_ctrl_frame_fwd
- send_xon

Packet Filtering

Applications control the packet-forwarding behaviors with packet filters.

The BNXT PMD supports hardware-based packet filtering:

- **UC (Unicast) MAC Filters**
  - No unicast packets are forwarded to an application except the one with DMAC address added to the port
  - At initialization, the station MAC address is added to the port
- **MC (Multicast) MAC Filters**
  - No multicast packets are forwarded to an application except the one with MC address added to the port
  - When the application listens to a multicast group, it adds the MC address to the port
- **VLAN Filtering Mode**
  - When enabled, no packets are forwarded to an application except the ones with the VLAN tag assigned to the port
- **Allmulticast Mode**
  - When enabled, every multicast packet received on the port is forwarded to the application
  - Typical usage is routing applications
- **Promiscuous Mode**
  - When enabled, every packet received on the port is forwarded to the application

Unicast MAC Filter

The application adds (or removes) MAC addresses to enable (or disable) whitelist filtering to accept packets.

```
testpmd> show port (port_id) macs
testpmd> mac_addr (add|remove) (port_id) (XX:XX:XX:XX:XX:XX)
```

Multicast MAC Filter

Application adds (or removes) Multicast addresses to enable (or disable) whitelist filtering to accept packets.

```
testpmd> show port (port_id) mcast_macs
testpmd> mcast_addr (add|remove) (port_id) (XX:XX:XX:XX:XX:XX)
```

Application adds (or removes) Multicast addresses to enable (or disable) whitelist filtering to accept packets.

Note that the BNXT PMD supports up to 16 MC MAC filters. if the user adds more than 16 MC MACs, the BNXT PMD puts the port into the Allmulticast mode.

VLAN Filtering

The application enables (or disables) VLAN filtering mode. When the mode is enabled, no packets are forwarded to an application except ones with VLAN tag assigned for the application.

```
testpmd> vlan set filter (on|off) (port_id)
testpmd> rx_vlan (add|rm) (vlan_id) (port_id)
```

Allmulticast Mode

The application enables (or disables) the allmulticast mode. When the mode is enabled, every multicast packet received is forwarded to the application.

```
testpmd> show port info (port_id)
testpmd> set allmulti (port_id) (on|off)
```

Promiscuous Mode

The application enables (or disables) the promiscuous mode. When the mode is enabled on a port, every packet received on the port is forwarded to the application.

```
testpmd> show port info (port_id)
testpmd> set promisc port_id (on|off)
```

Stateless Offloads

Like Linux, DPDK provides enabling hardware offload of some stateless processing (such as checksum calculation) of the stack, alleviating the CPU from having to burn cycles on every packet.

Listed below are the stateless offloads supported by the BNXT PMD:

- CRC offload (for both TX and RX packets)
- **Checksum Offload (for both TX and RX packets)**
  - IPv4 Checksum Offload
  - TCP Checksum Offload
  - UDP Checksum Offload
- **Segmentation/Reassembly Offloads**
  - TCP Segmentation Offload (TSO)
  - Large Receive Offload (LRO)
- **Multi-Queue**
  - Transmit Side Scaling (TSS)
  - Receive Side Scaling (RSS)

Also, the BNXT PMD supports stateless offloads on inner frames for tunneled packets. Listed below are the tunneling protocols supported by the BNXT PMD:

- VXLAN
- GRE
- NVGRE

Note that enabling (or disabling) stateless offloads requires applications to stop DPDK before changing configuration.

CRC Offload

The FCS (Frame Check Sequence) in the Ethernet frame is a four-octet CRC (Cyclic Redundancy Check) that allows detection of corrupted data within the entire frame as received on the receiver side.

The BNXT PMD supports hardware-based CRC offload:

- TX: calculate and insert CRC
- RX: check and remove CRC, notify the application on CRC error

Note that the CRC offload is always turned on.

Checksum Offload

The application enables hardware checksum calculation for IPv4, TCP, and UDP.

```
testpmd> port stop (port_id)
testpmd> csum set (ip|tcp|udp|outer-ip|outer-udp) (sw|hw) (port_id)
testpmd> set fwd csum
```

Multi-Queue

Multi-Queue, also known as TSS (Transmit Side Scaling) or RSS (Receive Side Scaling), is a common networking technique that allows for more efficient load balancing across multiple CPU cores.

The application enables multiple TX and RX queues when it is started.

```
testpmd -l 1,3,5 --master-lcore 1 --txq=2 -rxq=2 --nb-cores=2
```

TSS

TSS distributes network transmit processing across several hardware-based transmit queues, allowing outbound network traffic to be processed by multiple CPU cores.

RSS

RSS distributes network receive processing across several hardware-based receive queues, allowing inbound network traffic to be processed by multiple CPU cores.

The application can select the RSS mode, i.e. select the header fields that are included for hash calculation. The BNXT PMD supports the RSS mode of `default|ip|tcp|udp|none`, where default mode is L3 and L4.

For tunneled packets, RSS hash is calculated over inner frame header fields. Applications may want to select the tunnel header fields for hash calculation, and it will be supported in 20.08 using RSS level.

```
testpmd> port config (port_id) rss (all|default|ip|tcp|udp|none)

// note that the testpmd defaults the RSS mode to ip
// ensure to issue the command below to enable L4 header (TCP or UDP) along with IPv4 header
testpmd> port config (port_id) rss default

// to check the current RSS configuration, such as RSS function and RSS key
testpmd> show port (port_id) rss-hash key

// RSS is enabled by default. However, application can disable RSS as follows
testpmd> port config (port_id) rss none
```

Application can change the flow distribution, i.e. remap the received traffic to CPU cores, using RSS RETA (Redirection Table).

```
// application queries the current RSS RETA configuration
testpmd> show port (port_id) rss reta size (mask0, mask1)

// application changes the RSS RETA configuration
testpmd> port config (port_id) rss reta (hash, queue) [, (hash, queue)]
```

TSO

TSO (TCP Segmentation Offload), also known as LSO (Large Send Offload), enables the TCP/IP stack to pass to the NIC a larger datagram than the MTU (Maximum Transmit Unit). NIC breaks it into multiple segments before sending it to the network.

The BNXT PMD supports hardware-based TSO.

```
// display the status of TSO
testpmd> tso show (port_id)

// enable/disable TSO
testpmd> port config (port_id) tx_offload tcp_tso (on|off)

// set TSO segment size
testpmd> tso set segment_size (port_id)
```

The BNXT PMD also supports hardware-based tunneled TSO.

```
// display the status of tunneled TSO
testpmd> tunnel_tso show (port_id)

// enable/disable tunneled TSO
testpmd> port config (port_id) tx_offload vxlan_tnl_tso|gre_tnl_tso (on|off)

// set tunneled TSO segment size
testpmd> tunnel_tso set segment_size (port_id)
```

Note that the checksum offload is always assumed to be enabled for TSO.

LRO

LRO (Large Receive Offload) enables NIC to aggregate multiple incoming TCP/IP packets from a single stream into a larger buffer, before passing to the networking stack.

The BNXT PMD supports hardware-based LRO.

```
// display the status of LRO
testpmd> show port (port_id) rx_offload capabilities
testpmd> show port (port_id) rx_offload configuration

// enable/disable LRO
testpmd> port config (port_id) rx_offload tcp_lro (on|off)

// set max LRO packet (datagram) size
testpmd> port config (port_id) max-lro-pkt-size (max_size)
```

The BNXT PMD also supports tunneled LRO.

Some applications, such as routing, should *not* change the packet headers as they pass through (i.e. received from and sent back to the network). In such a case, GRO (Generic Receive Offload) should be used instead of LRO.

VLAN Insert/Strip

DPDK application offloads VLAN insert/strip to improve performance. The BNXT PMD supports hardware-based VLAN insert/strip offload for both single and double VLAN packets.

VLAN Insert

Application configures the VLAN TPID (Tag Protocol ID). By default, the TPID is 0x8100.

```
// configure outer TPID value for a port
testpmd> vlan set outer tpid (tpid_value) (port_id)
```

The inner TPID set will be rejected as the BNXT PMD supports inserting only an outer VLAN. Note that when a packet has a single VLAN, the tag is considered as outer, i.e. the inner VLAN is relevant only when a packet is double-tagged.

The BNXT PMD supports various TPID values shown below. Any other values will be rejected.

- `0x8100`
- `0x88a8`
- `0x9100`
- `0x9200`
- `0x9300`

The BNXT PMD supports the VLAN insert offload per-packet basis. The application provides the TCI (Tag Control Info) for a packet via mbuf. In turn, the BNXT PMD inserts the VLAN tag (via hardware) using the provided TCI along with the configured TPID.

```
// enable VLAN insert offload
testpmd> port config (port_id) rx_offload vlan_insert|qinq_insert (on|off)

if (mbuf->ol_flags && PKT_TX_QINQ)       // case-1: insert VLAN to single-tagged packet
    tci_value = mbuf->vlan_tci_outer
else if (mbuf->ol_flags && PKT_TX_VLAN)  // case-2: insert VLAN to untagged packet
    tci_value = mbuf->vlan_tci
```

VLAN Strip

The application configures the per-port VLAN strip offload.

```
// enable VLAN strip on a port
testpmd> port config (port_id) tx_offload vlan_strip (on|off)

// notify application VLAN strip via mbuf
mbuf->ol_flags |= PKT_RX_VLAN | PKT_RX_STRIPPED // outer VLAN is found and stripped
mbuf->vlan_tci = tci_value                      // TCI of the stripped VLAN
```

Time Synchronization

System operators may run a PTP (Precision Time Protocol) client application to synchronize the time on the NIC (and optionally, on the system) to a PTP master.

The BNXT PMD supports a PTP client application to communicate with a PTP master clock using DPDK IEEE1588 APIs. Note that the PTP client application needs to run on PF and vector mode needs to be disabled.

For the PTP time synchronization support, the BNXT PMD must be compiled with `CONFIG_RTE_LIBRTE_IEEE1588=y` (this compilation flag is currently pending).

```
testpmd> set fwd ieee1588 // enable IEEE 1588 mode
```

When enabled, the BNXT PMD configures hardware to insert IEEE 1588 timestamps to the outgoing PTP packets and reports IEEE 1588 timestamps from the incoming PTP packets to application via mbuf.

```
// RX packet completion will indicate whether the packet is PTP
mbuf->ol_flags |= PKT_RX_IEEE1588_PTP
```

Statistics Collection

In Linux, the *ethtool -S* enables us to query the NIC stats. DPDK provides the similar functionalities via rte_eth_stats and rte_eth_xstats.

The BNXT PMD supports both basic and extended stats collection:

- Basic stats
- Extended stats

Basic Stats

The application collects per-port and per-queue stats using rte_eth_stats APIs.

```
testpmd> show port stats (port_id)
```

Basic stats include:

- ipackets
- ibytes
- opackets
- obytes
- imissed
- ierrors
- oerrors

By default, per-queue stats for 16 queues are supported. For more than 16 queues, BNXT PMD should be compiled with `CONFIG_RTE_ETHDEV_QUEUE_STAT_CNTRS` set to the desired number of queues.

Extended Stats

Unlike basic stats, the extended stats are vendor-specific, i.e. each vendor provides its own set of counters.

The BNXT PMD provides a rich set of counters, including per-flow counters, per-cos counters, per-priority counters, etc.

```
testpmd> show port xstats (port_id)
```

Shown below is the elaborated sequence to retrieve extended stats:

```
// application queries the number of xstats
len = rte_eth_xstats_get(port_id, NULL, 0);
// BNXT PMD returns the size of xstats array (i.e. the number of entries)
// BNXT PMD returns 0, if the feature is compiled out or disabled

// application allocates memory for xstats
struct rte_eth_xstats_name *names; // name is 64 character or less
struct rte_eth_xstats *xstats;
names = calloc(len, sizeof(*names));
xstats = calloc(len, sizeof(*xstats));

// application retrieves xstats // names and values
ret = rte_eth_xstats_get_names(port_id, *names, len);
ret = rte_eth_xstats_get(port_id, *xstats, len);

// application checks the xstats
// application may repeat the below:
len = rte_eth_xstats_reset(port_id); // reset the xstats

// reset can be skipped, if application wants to see accumulated stats
// run traffic
// probably stop the traffic
// retrieve xstats // no need to retrieve xstats names again
// check xstats
```

Generic Flow Offload

Applications can get benefit by offloading all or part of flow processing to hardware. For example, applications can offload packet classification only (partial offload) or whole match-action (full offload).

DPDK offers the Generic Flow API (rte_flow API) to configure hardware to perform flow processing.

Listed below are the rte_flow APIs BNXT PMD supports:

- rte_flow_validate
- rte_flow_create
- rte_flow_destroy
- rte_flow_flush

Host Based Flow Table Management

Starting with 20.05 BNXT PMD supports host based flow table management. This is a new mechanism that should allow higher flow scalability than what is currently supported. This new approach also defines a new rte_flow parser, and mapper which currently supports basic packet classification in the receive path.

The feature uses a newly implemented control-plane firmware interface which optimizes flow insertions and deletions.

This is a tech preview feature, and is disabled by default. It can be enabled using bnxt devargs. For ex: "-w 0000:0d:00.0,host-based-truflow=1".

9.11.6 Application Support

Firmware

The BNXT PMD supports the application to retrieve the firmware version.

```
testpmd> show port info (port_id)
```

Note that the applications cannot update the firmware using BNXT PMD.

Multiple Processes

When two or more DPDK applications (e.g., testpmd and dpdk-pdump) share a single instance of DPDK, the BNXT PMD supports a single primary application and one or more secondary applications. Note that the DPDK-layer (not the PMD) ensures there is only one primary application.

There are two modes:

Manual mode

- Application notifies whether it is primary or secondary using *proc-type* flag
- 1st process should be spawned with `--proc-type=primary`
- All subsequent processes should be spawned with `--proc-type=secondary`

Auto detection mode

- Application is using `proc-type=auto` flag
- A process is spawned as a secondary if a primary is already running

The BNXT PMD uses the info to skip a device initialization, i.e. performs a device initialization only when being brought up by a primary application.

Runtime Queue Setup

Typically, a DPDK application allocates TX and RX queues statically: i.e. queues are allocated at start. However, an application may want to increase (or decrease) the number of queues dynamically for various reasons, e.g. power savings.

The BNXT PMD supports applications to increase or decrease queues at runtime.

```
testpmd> port config all (rxq|txq) (num_queues)
```

Note that a DPDK application must allocate default queues (one for TX and one for RX at minimum) at initialization.

Descriptor Status

Applications may use the descriptor status for various reasons, e.g. for power savings. For example, an application may stop polling and change to interrupt mode when the descriptor status shows no packets to service for a while.

The BNXT PMD supports the application to retrieve both TX and RX descriptor status.

```
testpmd> show port (port_id) (rxq|txq) (queue_id) desc (desc_id) status
```

Bonding

DPDK implements a light-weight library to allow PMDs to be bonded together and provide a single logical PMD to the application.

```
testpmd -l 0-3 -n4 --vdev 'net_bonding0,mode=0,slave=<PCI B:D.F device 1>,slave=<PCI B:D.F␣
↪device 2>,mac=XX:XX:XX:XX:XX:XX' - --socket_num=1 - -i --port-topology=chained
(ex) testpmd -l 1,3,5,7,9 -n4 --vdev 'net_bonding0,mode=0,slave=0000:82:00.0,slave=0000:82:00.
↪1,mac=00:1e:67:1d:fd:1d' - --socket-num=1 - -i --port-topology=chained
```

9.11.7 Vector Processing

Vector processing provides significantly improved performance over scalar processing (see Vector Processor, here).

The BNXT PMD supports the vector processing using SSE (Streaming SIMD Extensions) instructions on x86 platforms. The BNXT vPMD (vector mode PMD) is currently limited to Intel/AMD CPU architecture. Support for ARM is *not* currently implemented.

This improved performance comes from several optimizations:

- **Batching**
  - TX: processing completions in bulk

  - RX: allocating mbufs in bulk

- Chained mbufs are *not* supported, i.e. a packet should fit a single mbuf
- **Some stateless offloads are *not* supported with vector processing**
  - TX: no offloads will be supported
  - RX: reduced RX offloads (listed below) will be supported:

```
DEV_RX_OFFLOAD_VLAN_STRIP
DEV_RX_OFFLOAD_KEEP_CRC
DEV_RX_OFFLOAD_JUMBO_FRAME
DEV_RX_OFFLOAD_IPV4_CKSUM
DEV_RX_OFFLOAD_UDP_CKSUM
DEV_RX_OFFLOAD_TCP_CKSUM
DEV_RX_OFFLOAD_OUTER_IPV4_CKSUM
DEV_RX_OFFLOAD_RSS_HASH
DEV_RX_OFFLOAD_VLAN_FILTER
```

The BNXT Vector PMD is enabled in DPDK builds by default.

However, a decision to enable vector mode will be made when the port transitions from stopped to started. Any TX offloads or some RX offloads (other than listed above) will disable the vector mode. Offload configuration changes that impact vector mode must be made when the port is stopped.

Note that TX (or RX) vector mode can be enabled independently from RX (or TX) vector mode.

9.11.8 Appendix

Supported Chipsets and Adapters

BCM5730x NetXtreme-C® Family of Ethernet Network Controllers

Information about Ethernet adapters in the NetXtreme family of adapters can be found in the NetXtreme® Brand section of the Broadcom website.

- `M150c ... Single-port 40/50 Gigabit Ethernet Adapter`
- `P150c ... Single-port 40/50 Gigabit Ethernet Adapter`
- `P225c ... Dual-port 10/25 Gigabit Ethernet Adapter`

BCM574xx/575xx NetXtreme-E® Family of Ethernet Network Controllers

Information about Ethernet adapters in the NetXtreme family of adapters can be found in the NetXtreme® Brand section of the Broadcom website.

- `M125P .... Single-port OCP 2.0 10/25 Gigabit Ethernet Adapter`
- `M150P .... Single-port OCP 2.0 50 Gigabit Ethernet Adapter`
- `M150PM ... Single-port OCP 2.0 Multi-Host 50 Gigabit Ethernet Adapter`
- `M210P .... Dual-port OCP 2.0 10 Gigabit Ethernet Adapter`
- `M210TP ... Dual-port OCP 2.0 10 Gigabit Ethernet Adapter`
- `M1100G ... Single-port OCP 2.0 10/25/50/100 Gigabit Ethernet Adapter`

- N150G .... Single-port OCP 3.0 50 Gigabit Ethernet Adapter
- M225P .... Dual-port OCP 2.0 10/25 Gigabit Ethernet Adapter
- N210P .... Dual-port OCP 3.0 10 Gigabit Ethernet Adapter
- N210TP ... Dual-port OCP 3.0 10 Gigabit Ethernet Adapter
- N225P .... Dual-port OCP 3.0 10/25 Gigabit Ethernet Adapter
- N250G .... Dual-port OCP 3.0 50 Gigabit Ethernet Adapter
- N410SG ... Quad-port OCP 3.0 10 Gigabit Ethernet Adapter
- N410SGBT . Quad-port OCP 3.0 10 Gigabit Ethernet Adapter
- N425G .... Quad-port OCP 3.0 10/25 Gigabit Ethernet Adapter
- N1100G ... Single-port OCP 3.0 10/25/50/100 Gigabit Ethernet Adapter
- N2100G ... Dual-port OCP 3.0 10/25/50/100 Gigabit Ethernet Adapter
- N2200G ... Dual-port OCP 3.0 10/25/50/100/200 Gigabit Ethernet Adapter
- P150P .... Single-port 50 Gigabit Ethernet Adapter
- P210P .... Dual-port 10 Gigabit Ethernet Adapter
- P210TP ... Dual-port 10 Gigabit Ethernet Adapter
- P225P .... Dual-port 10/25 Gigabit Ethernet Adapter
- P410SG ... Quad-port 10 Gigabit Ethernet Adapter
- P410SGBT . Quad-port 10 Gigabit Ethernet Adapter
- P425G .... Quad-port 10/25 Gigabit Ethernet Adapter
- P1100G ... Single-port 10/25/50/100 Gigabit Ethernet Adapter
- P2100G ... Dual-port 10/25/50/100 Gigabit Ethernet Adapter
- P2200G ... Dual-port 10/25/50/100/200 Gigabit Ethernet Adapter

BCM588xx NetXtreme-S® Family of SmartNIC Network Controllers

Information about the Stingray family of SmartNIC adapters can be found in the Stingray® Brand section of the Broadcom website.

- PS225 ... Dual-port 25 Gigabit Ethernet SmartNIC

BCM5873x StrataGX® Family of Communications Processors

These ARM-based processors target a broad range of networking applications, including virtual CPE (vCPE) and NFV appliances, 10G service routers and gateways, control plane processing for Ethernet switches, and network-attached storage (NAS).

- StrataGX BCM58732 ... Octal-Core 3.0GHz 64-bit ARM®v8 Cortex®-A72 based SoC

9.12 CXGBE Poll Mode Driver

The CXGBE PMD (**librte_pmd_cxgbe**) provides poll mode driver support for **Chelsio Terminator** 10/25/40/100 Gbps family of adapters. CXGBE PMD has support for the latest Linux and FreeBSD operating systems.

CXGBEVF PMD provides poll mode driver support for SR-IOV Virtual functions and has support for the latest Linux operating systems.

More information can be found at Chelsio Communications Official Website.

9.12.1 Features

CXGBE and CXGBEVF PMD has support for:

- Multiple queues for TX and RX
- Receiver Side Steering (RSS) Receiver Side Steering (RSS) on IPv4, IPv6, IPv4-TCP/UDP, IPv6-TCP/UDP. For 4-tuple, enabling 'RSS on TCP' and 'RSS on TCP + UDP' is supported.
- VLAN filtering
- Checksum offload
- Promiscuous mode
- All multicast mode
- Port hardware statistics
- Jumbo frames
- Flow API - Support for both Wildcard (LE-TCAM) and Exact (HASH) match filters.

9.12.2 Limitations

The Chelsio Terminator series of devices provide two/four ports but expose a single PCI bus address, thus, librte_pmd_cxgbe registers itself as a PCI driver that allocates one Ethernet device per detected port.

For this reason, one cannot whitelist/blacklist a single port without whitelisting/blacklisting the other ports on the same device.

9.12.3 Supported Chelsio T5 NICs

- 1G NICs: T502-BT
- 10G NICs: T520-BT, T520-CR, T520-LL-CR, T520-SO-CR, T540-CR
- 40G NICs: T580-CR, T580-LP-CR, T580-SO-CR
- Other T5 NICs: T522-CR

9.12.4 Supported Chelsio T6 NICs

- 25G NICs: T6425-CR, T6225-CR, T6225-LL-CR, T6225-SO-CR
- 100G NICs: T62100-CR, T62100-LP-CR, T62100-SO-CR

9.12.5 Supported SR-IOV Chelsio NICs

SR-IOV virtual functions are supported on all the Chelsio NICs listed in *Supported Chelsio T5 NICs* and *Supported Chelsio T6 NICs*.

9.12.6 Prerequisites

- Requires firmware version **1.24.11.0** and higher. Visit Chelsio Download Center to get latest firmware bundled with the latest Chelsio Unified Wire package.

  For Linux, installing and loading the latest cxgb4 kernel driver from the Chelsio Unified Wire package should get you the latest firmware. More information can be obtained from the User Guide that is bundled with the Chelsio Unified Wire package.

  For FreeBSD, the latest firmware obtained from the Chelsio Unified Wire package must be manually flashed via cxgbetool available in FreeBSD source repository.

  Instructions on how to manually flash the firmware are given in section *Linux Installation* for Linux and section *FreeBSD Installation* for FreeBSD.

9.12.7 Pre-Installation Configuration

Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_CXGBE_PMD` (default **y**)

  Toggle compilation of librte_pmd_cxgbe driver.

  Note: This controls compilation of both CXGBE and CXGBEVF PMD.

Runtime Options

The following `devargs` options can be enabled at runtime. They must be passed as part of EAL arguments. For example,

```
testpmd -w 02:00.4,keep_ovlan=1 -- -i
```

Common Runtime Options

- `keep_ovlan` (default **0**)

  Toggle behavior to keep/strip outer VLAN in Q-in-Q packets. If enabled, the outer VLAN tag is preserved in Q-in-Q packets. Otherwise, the outer VLAN tag is stripped in Q-in-Q packets.

- `tx_mode_latency` (default **0**)

  When set to 1, Tx doesn't wait for max number of packets to get coalesced and sends the packets immediately at the end of the current Tx burst. When set to 0, Tx waits across multiple Tx bursts until the max number of packets have been coalesced. In this case, Tx only sends the coalesced packets to hardware once the max coalesce limit has been reached.

CXGBE VF Only Runtime Options

- `force_link_up` (default **0**)

  When set to 1, CXGBEVF PMD always forces link as up for all VFs on underlying Chelsio NICs. This enables multiple VFs on the same NIC to send traffic to each other even when the physical link is down.

CXGBE PF Only Runtime Options

- `filtermode` (default **0**)

  Apart from the 4-tuple (IP src/dst addresses and TCP/UDP src/dst port addresses), there are only 40-bits available to match other fields in packet headers. So, `filtermode` devarg allows user to dynamically select a 40-bit supported match field combination for LETCAM (wildcard) filters.

  Default value of **0** makes driver pick the combination configured in the firmware configuration file on the adapter.

  The supported flags and their corresponding values are shown in table below. These flags can be OR'd to create 1 of the multiple supported combinations for LETCAM filters.

FLAG	VALUE
Physical Port	0x1
PFVF	0x2
Destination MAC	0x4
Ethertype	0x8
Inner VLAN	0x10
Outer VLAN	0x20
IP TOS	0x40
IP Protocol	0x80

  The supported `filtermode` combinations and their corresponding OR'd values are shown in table below.

FILTERMODE COMBINATIONS	VALUE
Protocol, TOS, Outer VLAN, Port	0xE1
Protocol, TOS, Outer VLAN	0xE0
Protocol, TOS, Inner VLAN, Port	0xD1
Protocol, TOS, Inner VLAN	0xD0
Protocol, TOS, PFVF, Port	0xC3
Protocol, TOS, PFVF	0xC2
Protocol, TOS, Port	0xC1
Protocol, TOS	0xC0
Protocol, Outer VLAN, Port	0xA1
Protocol, Outer VLAN	0xA0
Protocol, Inner VLAN, Port	0x91
Protocol, Inner VLAN	0x90
Protocol, Ethertype, DstMAC, Port	0x8D
Protocol, Ethertype, DstMAC	0x8C
Protocol, Ethertype, Port	0x89
Protocol, Ethertype	0x88
Protocol, DstMAC, PFVF, Port	0x87
Protocol, DstMAC, PFVF	0x86
Protocol, DstMAC, Port	0x85
Protocol, DstMAC	0x84
Protocol, PFVF, Port	0x83
Protocol, PFVF	0x82
Protocol, Port	0x81
Protocol	0x80
TOS, Outer VLAN, Port	0x61
TOS, Outer VLAN	0x60
TOS, Inner VLAN, Port	0x51
TOS, Inner VLAN	0x50
TOS, Ethertype, DstMAC, Port	0x4D
TOS, Ethertype, DstMAC	0x4C
TOS, Ethertype, Port	0x49
TOS, Ethertype	0x48
TOS, DstMAC, PFVF, Port	0x47
TOS, DstMAC, PFVF	0x46
TOS, DstMAC, Port	0x45
TOS, DstMAC	0x44
TOS, PFVF, Port	0x43
TOS, PFVF	0x42
TOS, Port	0x41
TOS	0x40
Outer VLAN, Inner VLAN, Port	0x31
Outer VLAN, Ethertype, Port	0x29
Outer VLAN, Ethertype	0x28
Outer VLAN, DstMAC, Port	0x25
Outer VLAN, DstMAC	0x24
Outer VLAN, Port	0x21
Outer VLAN	0x20
Inner VLAN, Ethertype, Port	0x19

continues on next page

Table 9.2 – continued from previous page

FILTERMODE COMBINATIONS	VALUE
Inner VLAN, Ethertype	0x18
Inner VLAN, DstMAC, Port	0x15
Inner VLAN, DstMAC	0x14
Inner VLAN, Port	0x11
Inner VLAN	0x10
Ethertype, DstMAC, Port	0xD
Ethertype, DstMAC	0xC
Ethertype, PFVF, Port	0xB
Ethertype, PFVF	0xA
Ethertype, Port	0x9
Ethertype	0x8
DstMAC, PFVF, Port	0x7
DstMAC, PFVF	0x6
DstMAC, Port	0x5
Destination MAC	0x4
PFVF, Port	0x3
PFVF	0x2
Physical Port	0x1

For example, to enable matching `ethertype` field in Ethernet header, and `protocol` field in IPv4 header, the `filtermode` combination must be given as:

```
testpmd -w 02:00.4,filtermode=0x88 -- -i
```

- `filtermask` (default **0**)

  `filtermask` devarg works similar to `filtermode`, but is used to configure a filter mode combination for HASH (exact-match) filters.

  Note: The combination chosen for `filtermask` devarg **must be a subset** of the combination chosen for `filtermode` devarg.

  Default value of **0** makes driver pick the combination configured in the firmware configuration file on the adapter.

  Note that the filter rule will only be inserted in HASH region, if the rule contains **all** the fields specified in the `filtermask` combination. Otherwise, the filter rule will get inserted in LETCAM region.

  The same combination list explained in the tables in `filtermode` devarg section earlier applies for `filtermask` devarg, as well.

  For example, to enable matching only protocol field in IPv4 header, the `filtermask` combination must be given as:

```
testpmd -w 02:00.4,filtermode=0x88,filtermask=0x80 -- -i
```

9.12.8 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

9.12.9 Linux

Linux Installation

Steps to manually install the latest firmware from the downloaded Chelsio Unified Wire package for Linux operating system are as follows:

1. Load the kernel module:

```
modprobe cxgb4
```

2. Use ifconfig to get the interface name assigned to Chelsio card:

```
ifconfig -a | grep "00:07:43"
```

   Example output:

```
p1p1      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C0
p1p2      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C8
```

3. Install cxgbtool:

```
cd <path_to_uwire>/tools/cxgbtool
make install
```

4. Use cxgbtool to load the firmware config file onto the card:

```
cxgbtool p1p1 loadcfg <path_to_uwire>/src/network/firmware/t5-config.txt
```

5. Use cxgbtool to load the firmware image onto the card:

```
cxgbtool p1p1 loadfw <path_to_uwire>/src/network/firmware/t5fw-*.bin
```

6. Unload and reload the kernel module:

```
modprobe -r cxgb4
modprobe cxgb4
```

7. Verify with ethtool:

```
ethtool -i p1p1 | grep "firmware"
```

   Example output:

```
firmware-version: 1.24.11.0, TP 0.1.23.2
```

Running testpmd

This section demonstrates how to launch **testpmd** with Chelsio devices managed by librte_pmd_cxgbe in Linux operating system.

1. Load the kernel module:

```
modprobe cxgb4
```

2. Get the PCI bus addresses of the interfaces bound to cxgb4 driver:

```
dmesg | tail -2
```

   Example output:

```
cxgb4 0000:02:00.4 p1p1: renamed from eth0
cxgb4 0000:02:00.4 p1p2: renamed from eth1
```

   Note: Both the interfaces of a Chelsio 2-port adapter are bound to the same PCI bus address.

3. Unload the kernel module:

```
modprobe -ar cxgb4 csiostor
```

4. Running testpmd

   Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

   Note: Currently, CXGBE PMD only supports the binding of PF4 for Chelsio NICs.

   Example output:

```
[...]
EAL: PCI device 0000:02:00.4 on NUMA socket -1
EAL:   probe driver: 1425:5401 rte_cxgbe_pmd
EAL:   PCI memory mapped at 0x7fd7c0200000
EAL:   PCI memory mapped at 0x7fd77cdfd000
EAL:   PCI memory mapped at 0x7fd7c10b7000
PMD: rte_cxgbe_pmd: fw: 1.24.11.0, TP: 0.1.23.2
PMD: rte_cxgbe_pmd: Coming up as MASTER: Initializing adapter
Interactive-mode selected
Configuring Port 0 (socket 0)
Port 0: 00:07:43:2D:EA:C0
Configuring Port 1 (socket 0)
Port 1: 00:07:43:2D:EA:C8
Checking link statuses...
PMD: rte_cxgbe_pmd: Port0: passive DA port module inserted
PMD: rte_cxgbe_pmd: Port1: passive DA port module inserted
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

Note: Flow control pause TX/RX is disabled by default and can be enabled via testpmd. Refer section *Enable/Disable Flow Control* for more details.

Configuring SR-IOV Virtual Functions

This section demonstrates how to enable SR-IOV virtual functions on Chelsio NICs and demonstrates how to run testpmd with SR-IOV virtual functions.

1. Load the kernel module:

```
modprobe cxgb4
```

2. Get the PCI bus addresses of the interfaces bound to cxgb4 driver:

```
dmesg | tail -2
```

   Example output:

```
cxgb4 0000:02:00.4 p1p1: renamed from eth0
cxgb4 0000:02:00.4 p1p2: renamed from eth1
```

   Note: Both the interfaces of a Chelsio 2-port adapter are bound to the same PCI bus address.

3. Use ifconfig to get the interface name assigned to Chelsio card:

```
ifconfig -a | grep "00:07:43"
```

   Example output:

```
p1p1      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C0
p1p2      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C8
```

4. Bring up the interfaces:

```
ifconfig p1p1 up
ifconfig p1p2 up
```

5. Instantiate SR-IOV Virtual Functions. PF0..3 can be used for SR-IOV VFs. Multiple VFs can be instantiated on each of PF0..3. To instantiate one SR-IOV VF on each PF0 and PF1:

```
echo 1 > /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:02\:00.1/sriov_numvfs
```

6. Get the PCI bus addresses of the virtual functions:

```
lspci | grep -i "Chelsio" | grep -i "VF"
```

   Example output:

```
02:01.0 Ethernet controller: Chelsio Communications Inc T540-CR Unified Wire Ethernet␣
↪Controller [VF]
02:01.1 Ethernet controller: Chelsio Communications Inc T540-CR Unified Wire Ethernet␣
↪Controller [VF]
```

7. Running testpmd

   Follow instructions available in the document *compiling and testing a PMD for a NIC* to bind virtual functions and run testpmd.

   Example output:

```
[...]
EAL: PCI device 0000:02:01.0 on NUMA socket 0
EAL:   probe driver: 1425:5803 net_cxgbevf
PMD: rte_cxgbe_pmd: Firmware version: 1.24.11.0
PMD: rte_cxgbe_pmd: TP Microcode version: 0.1.23.2
PMD: rte_cxgbe_pmd: Chelsio rev 0
PMD: rte_cxgbe_pmd: No bootstrap loaded
PMD: rte_cxgbe_pmd: No Expansion ROM loaded
PMD: rte_cxgbe_pmd:  0000:02:01.0 Chelsio rev 0 1G/10GBASE-SFP
EAL: PCI device 0000:02:01.1 on NUMA socket 0
EAL:   probe driver: 1425:5803 net_cxgbevf
PMD: rte_cxgbe_pmd: Firmware version: 1.24.11.0
PMD: rte_cxgbe_pmd: TP Microcode version: 0.1.23.2
PMD: rte_cxgbe_pmd: Chelsio rev 0
PMD: rte_cxgbe_pmd: No bootstrap loaded
PMD: rte_cxgbe_pmd: No Expansion ROM loaded
PMD: rte_cxgbe_pmd:  0000:02:01.1 Chelsio rev 0 1G/10GBASE-SFP
Configuring Port 0 (socket 0)
Port 0: 06:44:29:44:40:00
Configuring Port 1 (socket 0)
Port 1: 06:44:29:44:40:10
Checking link statuses...
Done
testpmd>
```

9.12.10 FreeBSD

FreeBSD Installation

Steps to manually install the latest firmware from the downloaded Chelsio Unified Wire package for FreeBSD operating system are as follows:

1. Load the kernel module:

```
kldload if_cxgbe
```

2. Use dmesg to get the t5nex instance assigned to the Chelsio card:

```
dmesg | grep "t5nex"
```

   Example output:

```
t5nex0: <Chelsio T520-CR> irq 16 at device 0.4 on pci2
cxl0: <port 0> on t5nex0
cxl1: <port 1> on t5nex0
t5nex0: PCIe x8, 2 ports, 14 MSI-X interrupts, 31 eq, 13 iq
```

   In the example above, a Chelsio T520-CR card is bound to a t5nex0 instance.

3. Install cxgbetool from FreeBSD source repository:

```
cd <path_to_FreeBSD_source>/tools/tools/cxgbetool/
make && make install
```

4. Use cxgbetool to load the firmware image onto the card:

```
cxgbetool t5nex0 loadfw <path_to_uwire>/src/network/firmware/t5fw-*.bin
```

5. Unload and reload the kernel module:

```
kldunload if_cxgbe
kldload if_cxgbe
```

6. Verify with sysctl:

```
sysctl -a | grep "t5nex" | grep "firmware"
```

   Example output:

```
dev.t5nex.0.firmware_version: 1.24.11.0
```

Running testpmd

This section demonstrates how to launch **testpmd** with Chelsio devices managed by librte_pmd_cxgbe in FreeBSD operating system.

1. Change to DPDK source directory where the target has been compiled in section *Driver compilation and testing*:

```
cd <DPDK-source-directory>
```

2. Copy the contigmem kernel module to /boot/kernel directory:

```
cp x86_64-native-freebsd-clang/kmod/contigmem.ko /boot/kernel/
```

3. Add the following lines to /boot/loader.conf:

```
# reserve 2 x 1G blocks of contiguous memory using contigmem driver
hw.contigmem.num_buffers=2
hw.contigmem.buffer_size=1073741824
# load contigmem module during boot process
contigmem_load="YES"
```

   The above lines load the contigmem kernel module during boot process and allocate 2 x 1G blocks of contiguous memory to be used for DPDK later on. This is to avoid issues with potential memory fragmentation during later system up time, which may result in failure of allocating the contiguous memory required for the contigmem kernel module.

4. Restart the system and ensure the contigmem module is loaded successfully:

```
reboot
kldstat | grep "contigmem"
```

   Example output:

```
2    1 0xffffffff817f1000 3118     contigmem.ko
```

5. Repeat step 1 to ensure that you are in the DPDK source directory.

6. Load the cxgbe kernel module:

```
kldload if_cxgbe
```

7. Get the PCI bus addresses of the interfaces bound to t5nex driver:

```
pciconf -l | grep "t5nex"
```

Example output:

```
t5nex0@pci0:2:0:4: class=0x020000 card=0x00001425 chip=0x54011425 rev=0x00
```

In the above example, the t5nex0 is bound to 2:0:4 bus address.

Note: Both the interfaces of a Chelsio 2-port adapter are bound to the same PCI bus address.

8. Unload the kernel module:

```
kldunload if_cxgbe
```

9. Set the PCI bus addresses to hw.nic_uio.bdfs kernel environment parameter:

```
kenv hw.nic_uio.bdfs="2:0:4"
```

This automatically binds 2:0:4 to nic_uio kernel driver when it is loaded in the next step.

Note: Currently, CXGBE PMD only supports the binding of PF4 for Chelsio NICs.

10. Load nic_uio kernel driver:

```
kldload ./x86_64-native-freebsd-clang/kmod/nic_uio.ko
```

11. Start testpmd with basic parameters:

```
./x86_64-native-freebsd-clang/app/testpmd -l 0-3 -n 4 -w 0000:02:00.4 -- -i
```

Example output:

```
[...]
EAL: PCI device 0000:02:00.4 on NUMA socket 0
EAL:   probe driver: 1425:5401 rte_cxgbe_pmd
EAL:   PCI memory mapped at 0x8007ec000
EAL:   PCI memory mapped at 0x842800000
EAL:   PCI memory mapped at 0x80086c000
PMD: rte_cxgbe_pmd: fw: 1.24.11.0, TP: 0.1.23.2
PMD: rte_cxgbe_pmd: Coming up as MASTER: Initializing adapter
Interactive-mode selected
Configuring Port 0 (socket 0)
Port 0: 00:07:43:2D:EA:C0
Configuring Port 1 (socket 0)
Port 1: 00:07:43:2D:EA:C8
Checking link statuses...
PMD: rte_cxgbe_pmd: Port0: passive DA port module inserted
```

(continues on next page)

(continued from previous page)

```
PMD: rte_cxgbe_pmd: Port1: passive DA port module inserted
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

Note: Flow control pause TX/RX is disabled by default and can be enabled via testpmd. Refer section *Enable/Disable Flow Control* for more details.

9.12.11 Sample Application Notes

Enable/Disable Flow Control

Flow control pause TX/RX is disabled by default and can be enabled via testpmd as follows:

```
testpmd> set flow_ctrl rx on tx on 0 0 0 0 mac_ctrl_frame_fwd off autoneg on 0
testpmd> set flow_ctrl rx on tx on 0 0 0 0 mac_ctrl_frame_fwd off autoneg on 1
```

To disable again, run:

```
testpmd> set flow_ctrl rx off tx off 0 0 0 0 mac_ctrl_frame_fwd off autoneg off 0
testpmd> set flow_ctrl rx off tx off 0 0 0 0 mac_ctrl_frame_fwd off autoneg off 1
```

Jumbo Mode

There are two ways to enable sending and receiving of jumbo frames via testpmd. One method involves using the **mtu** command, which changes the mtu of an individual port without having to stop the selected port. Another method involves stopping all the ports first and then running **max-pkt-len** command to configure the mtu of all the ports with a single command.

- To configure each port individually, run the mtu command as follows:

```
testpmd> port config mtu 0 9000
testpmd> port config mtu 1 9000
```

- To configure all the ports at once, stop all the ports first and run the max-pkt-len command as follows:

```
testpmd> port stop all
testpmd> port config all max-pkt-len 9000
```

9.13 DPAA Poll Mode Driver

The DPAA NIC PMD (**librte_pmd_dpaa**) provides poll mode driver support for the inbuilt NIC found in the **NXP DPAA** SoC family.

More information can be found at NXP Official Website.

9.13.1 NXP DPAA (Data Path Acceleration Architecture - Gen 1)

This section provides an overview of the NXP DPAA architecture and how it is integrated into the DPDK.

Contents summary

- DPAA overview
- DPAA driver architecture overview

DPAA Overview

Reference: FSL DPAA Architecture.

The QorIQ Data Path Acceleration Architecture (DPAA) is a set of hardware components on specific QorIQ series multicore processors. This architecture provides the infrastructure to support simplified sharing of networking interfaces and accelerators by multiple CPU cores, and the accelerators themselves.

DPAA includes:

- Cores
- Network and packet I/O
- Hardware offload accelerators
- Infrastructure required to facilitate flow of packets between the components above

Infrastructure components are:

- The Queue Manager (QMan) is a hardware accelerator that manages frame queues. It allows CPUs and other accelerators connected to the SoC datapath to enqueue and dequeue ethernet frames, thus providing the infrastructure for data exchange among CPUs and datapath accelerators.
- The Buffer Manager (BMan) is a hardware buffer pool management block that allows software and accelerators on the datapath to acquire and release buffers in order to build frames.

Hardware accelerators are:

- SEC - Cryptographic accelerator
- PME - Pattern matching engine

The Network and packet I/O component:

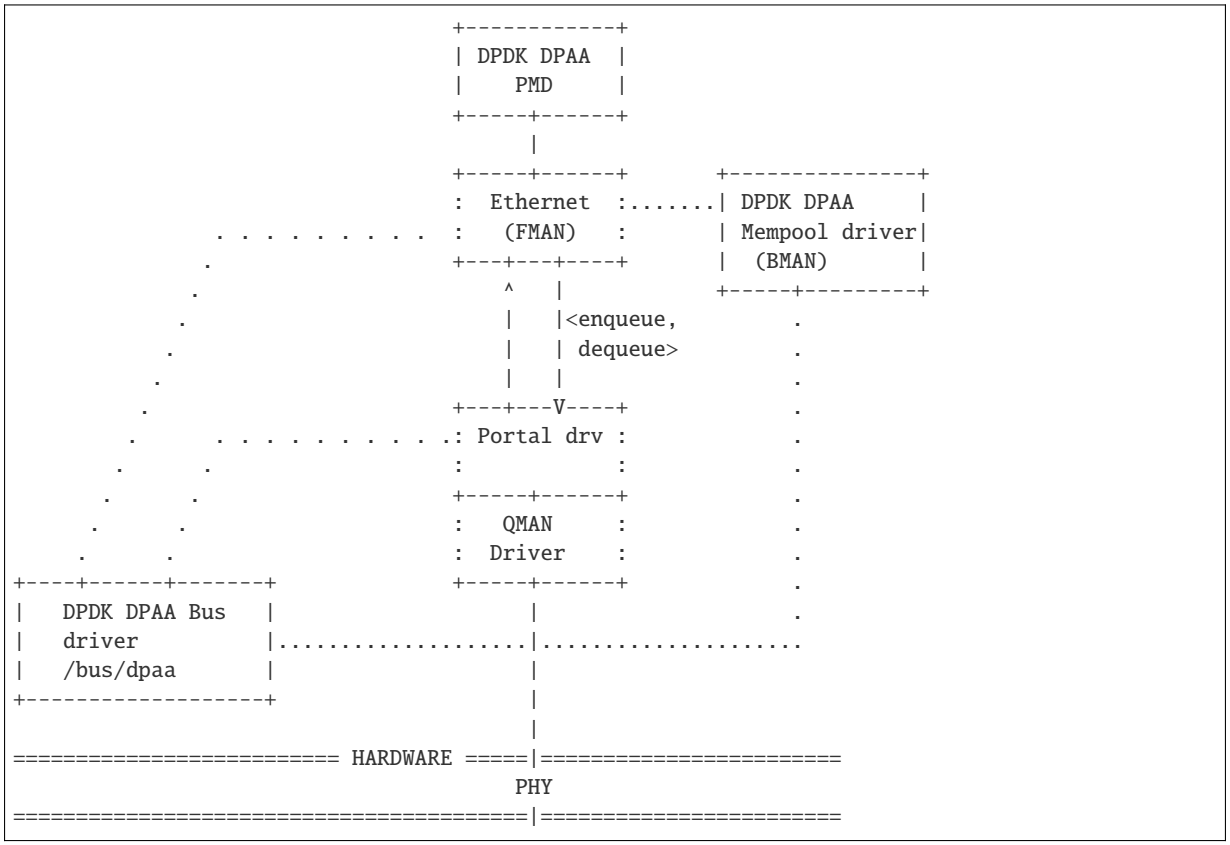
- The Frame Manager (FMan) is a key component in the DPAA and makes use of the DPAA infrastructure (QMan and BMan). FMan is responsible for packet distribution and policing. Each frame can be parsed, classified and results may be attached to the frame. This meta data can be used to select particular QMan queue, which the packet is forwarded to.

9.13.2 DPAA DPDK - Poll Mode Driver Overview

This section provides an overview of the drivers for DPAA:

- Bus driver and associated "DPAA infrastructure" drivers
- Functional object drivers (such as Ethernet).

Brief description of each driver is provided in layout below as well as in the following sections.



In the above representation, solid lines represent components which interface with DPDK RTE Framework and dotted lines represent DPAA internal components.

DPAA Bus driver

The DPAA bus driver is a `rte_bus` driver which scans the platform like bus. Key functions include:

- Scanning and parsing the various objects and adding them to their respective device list.
- Performing probe for available drivers against each scanned device
- Creating necessary ethernet instance before passing control to the PMD

DPAA NIC Driver (PMD)

DPAA PMD is traditional DPDK PMD which provides necessary interface between RTE framework and DPAA internal components/drivers.

- Once devices have been identified by DPAA Bus, each device is associated with the PMD
- PMD is responsible for implementing necessary glue layer between RTE APIs and lower level QMan and FMan blocks. The Ethernet driver is bound to a FMAN port and implements the interfaces needed to connect the DPAA network interface to the network stack. Each FMAN Port corresponds to a DPDK network interface.

Features

Features of the DPAA PMD are:

- Multiple queues for TX and RX
- Receive Side Scaling (RSS)
- Packet type information
- Checksum offload
- Promiscuous mode

DPAA Mempool Driver

DPAA has a hardware offloaded buffer pool manager, called BMan, or Buffer Manager.

- Using standard Mempools operations RTE API, the mempool driver interfaces with RTE to service each mempool creation, deletion, buffer allocation and deallocation requests.
- Each FMAN instance has a BMan pool attached to it during initialization. Each Tx frame can be automatically released by hardware, if allocated from this pool.

9.13.3 Whitelisting & Blacklisting

For blacklisting a DPAA device, following commands can be used.

```
<dpdk app> <EAL args> -b "dpaa_bus:fmX-macY" -- ...
e.g. "dpaa_bus:fm1-mac4"
```

9.13.4 Supported DPAA SoCs

- LS1043A/LS1023A
- LS1046A/LS1026A

9.13.5 Prerequisites

See *NXP QorIQ DPAA Board Support Package* for setup information

- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

Note: Some part of dpaa bus code (qbman and fman - library) routines are dual licensed (BSD & GPLv2), however they are used as BSD in DPDK in userspace.

9.13.6 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_DPAA_BUS` (default `y`)

  Toggle compilation of the `librte_bus_dpaa` driver.

- `CONFIG_RTE_LIBRTE_DPAA_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_dpaa` driver.

- `CONFIG_RTE_LIBRTE_DPAA_DEBUG_DRIVER` (default `n`)

  Toggles display of bus configurations and enables a debugging queue to fetch error (Rx/Tx) packets to driver. By default, packets with errors (like wrong checksum) are dropped by the hardware.

- `CONFIG_RTE_LIBRTE_DPAA_HWDEBUG` (default `n`)

  Enables debugging of the Queue and Buffer Manager layer which interacts with the DPAA hardware.

Environment Variables

DPAA drivers uses the following environment variables to configure its state during application initialization:

- `DPAA_NUM_RX_QUEUES` (default 1)

  This defines the number of Rx queues configured for an application, per port. Hardware would distribute across these many number of queues on Rx of packets. In case the application is configured to use lesser number of queues than configured above, it might result in packet loss (because of distribution).

- `DPAA_PUSH_QUEUES_NUMBER` (default 4)

  This defines the number of High performance queues to be used for ethdev Rx. These queues use one private HW portal per queue configured, so they are limited in the system. The first configured ethdev queues will be automatically be assigned from the these high perf PUSH queues. Any queue configuration beyond that will be standard Rx queues. The application can choose to change their number if HW portals are limited. The valid values are from '0' to '4'. The values shall be set to '0' if the application want to use eventdev with DPAA device. Currently these queues are not used for LS1023/LS1043 platform by default.

9.13.7 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

1. Running testpmd:

   Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

   Example output:

```
./arm64-dpaa-linux-gcc/testpmd -c 0xff -n 1 \
  -- -i --portmask=0x3 --nb-cores=1 --no-flush-rx

.....
EAL: Registered [pci] bus.
EAL: Registered [dpaa] bus.
EAL: Detected 4 lcore(s)
.....
EAL: dpaa: Bus scan completed
.....
Configuring Port 0 (socket 0)
Port 0: 00:00:00:00:00:01
Configuring Port 1 (socket 0)
Port 1: 00:00:00:00:00:02
.....
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

9.13.8 Limitations

Platform Requirement

DPAA drivers for DPDK can only work on NXP SoCs as listed in the `Supported DPAA SoCs`.

Maximum packet length

The DPAA SoC family support a maximum of a 10240 jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 10240, frames up to 10240 bytes can still reach the host interface.

Multiprocess Support

Current version of DPAA driver doesn't support multi-process applications where I/O is performed using secondary processes. This feature would be implemented in subsequent versions.

9.14 DPAA2 Poll Mode Driver

The DPAA2 NIC PMD (**librte_pmd_dpaa2**) provides poll mode driver support for the inbuilt NIC found in the **NXP DPAA2** SoC family.

More information can be found at NXP Official Website.

9.14.1 NXP DPAA2 (Data Path Acceleration Architecture Gen2)

This section provides an overview of the NXP DPAA2 architecture and how it is integrated into the DPDK.

Contents summary

- DPAA2 overview
- Overview of DPAA2 objects
- DPAA2 driver architecture overview

DPAA2 Overview

Reference: FSL MC BUS in Linux Kernel.

DPAA2 is a hardware architecture designed for high-speed network packet processing. DPAA2 consists of sophisticated mechanisms for processing Ethernet packets, queue management, buffer management, autonomous L2 switching, virtual Ethernet bridging, and accelerator (e.g. crypto) sharing.

A DPAA2 hardware component called the Management Complex (or MC) manages the DPAA2 hardware resources. The MC provides an object-based abstraction for software drivers to use the DPAA2 hardware.

The MC uses DPAA2 hardware resources such as queues, buffer pools, and network ports to create functional objects/devices such as network interfaces, an L2 switch, or accelerator instances.

The MC provides memory-mapped I/O command interfaces (MC portals) which DPAA2 software drivers use to operate on DPAA2 objects:

The diagram below shows an overview of the DPAA2 resource management architecture:

```
+--------------------------------------+
|                  OS                  |
|                        DPAA2 drivers |
|                             |        |
+-----------------------------|--------+
                              |
                              | (create,discover,connect
                              |  config,use,destroy)
                              |
                DPAA2         |
+-----------------------| mc portal |-+
|                             |        |
|   +- - - - - - - - - - - -V- - -+  |
|   |                              |  |
|   |   Management Complex (MC)    |  |
|   |                              |  |
|   +- - - - - - - - - - - - - - -+  |
|                                      |
| Hardware                  Hardware   |
```

(continues on next page)

(continued from previous page)

```
| Resources                 Objects    |
| ---------                 -------    |
| -queues                   -DPRC      |
| -buffer pools             -DPMCP     |
| -Eth MACs/ports           -DPIO      |
| -network interface        -DPNI      |
|  profiles                 -DPMAC     |
| -queue portals            -DPBP      |
| -MC portals                ...       |
|  ...                                 |
|                                      |
+--------------------------------------+
```

The MC mediates operations such as create, discover, connect, configuration, and destroy. Fast-path operations on data, such as packet transmit/receive, are not mediated by the MC and are done directly using memory mapped regions in DPIO objects.

Overview of DPAA2 Objects

The section provides a brief overview of some key DPAA2 objects. A simple scenario is described illustrating the objects involved in creating a network interfaces.

DPRC (Datapath Resource Container)

> A DPRC is a container object that holds all the other types of DPAA2 objects. In the example diagram below there are 8 objects of 5 types (DPMCP, DPIO, DPBP, DPNI, and DPMAC) in the container.

```
+-------------------------------------------------------+
| DPRC                                                  |
|                                                       |
|  +-------+  +-------+  +-------+  +-------+  +-------+  |
|  | DPMCP |  | DPIO  |  | DPBP  |  | DPNI  |  | DPMAC |  |
|  +-------+  +-------+  +-------+  +---+---+  +---+---+  |
|  | DPMCP |  | DPIO  |                                  |
|  +-------+  +-------+                                  |
|  | DPMCP |                                            |
|  +-------+                                            |
|                                                       |
+-------------------------------------------------------+
```

From the point of view of an OS, a DPRC behaves similar to a plug and play bus, like PCI. DPRC commands can be used to enumerate the contents of the DPRC, discover the hardware objects present (including mappable regions and interrupts).

```
DPRC.1 (bus)
  |
  +--+--------+-------+-------+-------+
     |        |       |       |       |
   DPMCP.1  DPIO.1  DPBP.1  DPNI.1  DPMAC.1
   DPMCP.2  DPIO.2
   DPMCP.3
```

Hardware objects can be created and destroyed dynamically, providing the ability to hot plug/unplug objects in and out of the DPRC.

A DPRC has a mappable MMIO region (an MC portal) that can be used to send MC commands. It has an interrupt for status events (like hotplug).

All objects in a container share the same hardware "isolation context". This means that with respect to an IOMMU the isolation granularity is at the DPRC (container) level, not at the individual object level.

DPRCs can be defined statically and populated with objects via a config file passed to the MC when firmware starts it. There is also a Linux user space tool called "restool" that can be used to create/destroy containers and objects dynamically.

DPAA2 Objects for an Ethernet Network Interface

A typical Ethernet NIC is monolithic– the NIC device contains TX/RX queuing mechanisms, configuration mechanisms, buffer management, physical ports, and interrupts. DPAA2 uses a more granular approach utilizing multiple hardware objects. Each object provides specialized functions. Groups of these objects are used by software to provide Ethernet network interface functionality. This approach provides efficient use of finite hardware resources, flexibility, and performance advantages.

The diagram below shows the objects needed for a simple network interface configuration on a system with 2 CPUs.

Below the objects are described. For each object a brief description is provided along with a summary of the kinds of operations the object supports and a summary of key resources of the object (MMIO regions and IRQs).

DPMAC (Datapath Ethernet MAC): represents an Ethernet MAC, a hardware device that connects to an Ethernet PHY and allows physical transmission and reception of Ethernet frames.

- MMIO regions: none
- IRQs: DPNI link change
- commands: set link up/down, link config, get stats, IRQ config, enable, reset

DPNI (Datapath Network Interface): contains TX/RX queues, network interface configuration, and RX buffer pool configuration mechanisms. The TX/RX queues are in memory and are identified by queue number.

- MMIO regions: none
- IRQs: link state
- commands: port config, offload config, queue config, parse/classify config, IRQ config, enable, reset

DPIO (Datapath I/O): provides interfaces to enqueue and dequeue packets and do hardware buffer pool management operations. The DPAA2 architecture separates the mechanism to access queues (the DPIO object) from the queues themselves. The DPIO provides an MMIO interface to enqueue/dequeue packets. To enqueue something a descriptor is written to the DPIO MMIO region, which includes the target queue number. There will typically be one DPIO assigned to each CPU. This allows all CPUs to simultaneously perform enqueue/dequeued operations. DPIOs are expected to be shared by different DPAA2 drivers.

- MMIO regions: queue operations, buffer management
- IRQs: data availability, congestion notification, buffer pool depletion
- commands: IRQ config, enable, reset

DPBP (Datapath Buffer Pool): represents a hardware buffer pool.

- MMIO regions: none
- IRQs: none
- commands: enable, reset

DPMCP (Datapath MC Portal): provides an MC command portal. Used by drivers to send commands to the MC to manage objects.

- MMIO regions: MC command portal
- IRQs: command completion
- commands: IRQ config, enable, reset

Object Connections

Some objects have explicit relationships that must be configured:

- DPNI <–> DPMAC
- DPNI <–> DPNI
- DPNI <–> L2-switch-port

A DPNI must be connected to something such as a DPMAC, another DPNI, or L2 switch port. The DPNI connection is made via a DPRC command.

- DPNI <–> DPBP

A network interface requires a 'buffer pool' (DPBP object) which provides a list of pointers to memory where received Ethernet data is to be copied. The Ethernet driver configures the DPBPs associated with the network interface.

Interrupts

All interrupts generated by DPAA2 objects are message interrupts. At the hardware level message interrupts generated by devices will normally have 3 components– 1) a non-spoofable 'device-id' expressed on the hardware bus, 2) an address, 3) a data value.

In the case of DPAA2 devices/objects, all objects in the same container/DPRC share the same 'device-id'. For ARM-based SoC this is the same as the stream ID.

9.14.2 DPAA2 DPDK - Poll Mode Driver Overview

This section provides an overview of the drivers for DPAA2– 1) the bus driver and associated "DPAA2 infrastructure" drivers and 2) functional object drivers (such as Ethernet).

As described previously, a DPRC is a container that holds the other types of DPAA2 objects. It is functionally similar to a plug-and-play bus controller.

Each object in the DPRC is a Linux "device" and is bound to a driver. The diagram below shows the dpaa2 drivers involved in a networking scenario and the objects bound to each driver. A brief description of each driver follows.

A brief description of each driver is provided below.

DPAA2 bus driver

The DPAA2 bus driver is a rte_bus driver which scans the fsl-mc bus. Key functions include:

- Reading the container and setting up vfio group
- Scanning and parsing the various MC objects and adding them to their respective device list.

Additionally, it also provides the object driver for generic MC objects.

DPIO driver

The DPIO driver is bound to DPIO objects and provides services that allow other drivers such as the Ethernet driver to enqueue and dequeue data for their respective objects. Key services include:

- Data availability notifications
- Hardware queuing operations (enqueue and dequeue of data)
- Hardware buffer pool management

To transmit a packet the Ethernet driver puts data on a queue and invokes a DPIO API. For receive, the Ethernet driver registers a data availability notification callback. To dequeue a packet a DPIO API is used.

There is typically one DPIO object per physical CPU for optimum performance, allowing different CPUs to simultaneously enqueue and dequeue data.

The DPIO driver operates on behalf of all DPAA2 drivers active – Ethernet, crypto, compression, etc.

DPBP based Mempool driver

The DPBP driver is bound to a DPBP objects and provides services to create a hardware offloaded packet buffer mempool.

DPAA2 NIC Driver

The Ethernet driver is bound to a DPNI and implements the kernel interfaces needed to connect the DPAA2 network interface to the network stack.

Each DPNI corresponds to a DPDK network interface.

Features

Features of the DPAA2 PMD are:

- Multiple queues for TX and RX
- Receive Side Scaling (RSS)
- MAC/VLAN filtering
- Packet type information
- Checksum offload
- Promiscuous mode
- Multicast mode
- Port hardware statistics
- Jumbo frames
- Link flow control
- Scattered and gather for TX and RX

9.14.3 Supported DPAA2 SoCs

- LX2160A
- LS2084A/LS2044A
- LS2088A/LS2048A
- LS1088A/LS1048A

9.14.4 Prerequisites

See *NXP QorIQ DPAA2 Board Support Package* for setup information

Currently supported by DPDK:

- NXP LSDK **19.08+**.
- MC Firmware version **10.18.0** and higher.
- Supported architectures: **arm64 LE**.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

Note: Some part of fslmc bus code (mc flib - object library) routines are dual licensed (BSD & GPLv2), however they are used as BSD in DPDK in userspace.

9.14.5 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_FSLMC_BUS` (default `y`)

  Toggle compilation of the `librte_bus_fslmc` driver.

- `CONFIG_RTE_LIBRTE_DPAA2_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_dpaa2` driver.

- `CONFIG_RTE_LIBRTE_DPAA2_DEBUG_DRIVER` (default `n`)

  Toggle display of debugging messages/logic

- `CONFIG_RTE_LIBRTE_DPAA2_USE_PHYS_IOVA` (default `n`)

  Toggle to use physical address vs virtual address for hardware accelerators.

9.14.6 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

1. Running testpmd:

   Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

   Example output:

```
./testpmd -c 0xff -n 1 -- -i --portmask=0x3 --nb-cores=1 --no-flush-rx

.....
EAL: Registered [pci] bus.
EAL: Registered [fslmc] bus.
EAL: Detected 8 lcore(s)
```

(continues on next page)

(continued from previous page)

```
EAL: Probing VFIO support...
EAL: VFIO support initialized
.....
PMD: DPAA2: Processing Container = dprc.2
EAL: fslmc: DPRC contains = 51 devices
EAL: fslmc: Bus scan completed
.....
Configuring Port 0 (socket 0)
Port 0: 00:00:00:00:00:01
Configuring Port 1 (socket 0)
Port 1: 00:00:00:00:00:02
.....
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

- Use dev arg option `drv_loopback=1` to loopback packets at driver level. Any packet received will be reflected back by the driver on same port. e.g. `fslmc:dpni.1,drv_loopback=1`
- Use dev arg option `drv_no_prefetch=1` to disable prefetching of the packet pull command which is issued in the previous cycle. e.g. `fslmc:dpni.1,drv_no_prefetch=1`

9.14.7 Enabling logs

For enabling logging for DPAA2 PMD, following log-level prefix can be used:

```
<dpdk app> <EAL args> --log-level=bus.fslmc:<level> -- ...
```

Using `bus.fslmc` as log matching criteria, all FSLMC bus logs can be enabled which are lower than logging `level`.

Or

```
<dpdk app> <EAL args> --log-level=pmd.net.dpaa2:<level> -- ...
```

Using `pmd.net.dpaa2` as log matching criteria, all PMD logs can be enabled which are lower than logging `level`.

9.14.8 Whitelisting & Blacklisting

For blacklisting a DPAA2 device, following commands can be used.

```
<dpdk app> <EAL args> -b "fslmc:dpni.x" -- ...
```

Where x is the device object id as configured in resource container.

9.14.9 Limitations

Platform Requirement

DPAA2 drivers for DPDK can only work on NXP SoCs as listed in the `Supported DPAA2 SoCs`.

Maximum packet length

The DPAA2 SoC family support a maximum of a 10240 jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 10240, frames up to 10240 bytes can still reach the host interface.

Other Limitations

- RSS hash key cannot be modified.
- RSS RETA cannot be configured.

9.15 Driver for VM Emulated Devices

The DPDK EM poll mode driver supports the following emulated devices:

- qemu-kvm emulated Intel® 82540EM Gigabit Ethernet Controller (qemu e1000 device)
- VMware* emulated Intel® 82545EM Gigabit Ethernet Controller
- VMware emulated Intel® 8274L Gigabit Ethernet Controller.

9.15.1 Validated Hypervisors

The validated hypervisors are:

- KVM (Kernel Virtual Machine) with Qemu, version 0.14.0
- KVM (Kernel Virtual Machine) with Qemu, version 0.15.1
- VMware ESXi 5.0, Update 1

9.15.2 Recommended Guest Operating System in Virtual Machine

The recommended guest operating system in a virtualized environment is:

- Fedora* 18 (64-bit)

For supported kernel versions, refer to the *DPDK Release Notes*.

9.15.3 Setting Up a KVM Virtual Machine

The following describes a target environment:

- Host Operating System: Fedora 14
- Hypervisor: KVM (Kernel Virtual Machine) with Qemu version, 0.14.0
- Guest Operating System: Fedora 14
- Linux Kernel Version: Refer to the DPDK Getting Started Guide
- Target Applications: testpmd

The setup procedure is as follows:

1. Download qemu-kvm-0.14.0 from http://sourceforge.net/projects/kvm/files/qemu-kvm/ and install it in the Host OS using the following steps:

   When using a recent kernel (2.6.25+) with kvm modules included:

   ```
   tar xzf qemu-kvm-release.tar.gz cd qemu-kvm-release
   ./configure --prefix=/usr/local/kvm
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   When using an older kernel or a kernel from a distribution without the kvm modules, you must download (from the same link), compile and install the modules yourself:

   ```
   tar xjf kvm-kmod-release.tar.bz2
   cd kvm-kmod-release
   ./configure
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   Note that qemu-kvm installs in the /usr/local/bin directory.

   For more details about KVM configuration and usage, please refer to: http://www.linux-kvm.org/page/HOWTO1.

2. Create a Virtual Machine and install Fedora 14 on the Virtual Machine. This is referred to as the Guest Operating System (Guest OS).

3. Start the Virtual Machine with at least one emulated e1000 device.

   Note: The Qemu provides several choices for the emulated network device backend. Most commonly used is a TAP networking backend that uses a TAP networking device in the host. For more information about Qemu supported networking backends and different options for configuring networking at Qemu, please refer to:

   — http://www.linux-kvm.org/page/Networking

   — http://wiki.qemu.org/Documentation/Networking

   — http://qemu.weilnetz.de/qemu-doc.html

   For example, to start a VM with two emulated e1000 devices, issue the following command:

```
/usr/local/kvm/bin/qemu-system-x86_64 -cpu host -smp 4 -hda qemu1.raw -m 1024
-net nic,model=e1000,vlan=1,macaddr=DE:AD:1E:00:00:01
-net tap,vlan=1,ifname=tapvm01,script=no,downscript=no
-net nic,model=e1000,vlan=2,macaddr=DE:AD:1E:00:00:02
-net tap,vlan=2,ifname=tapvm02,script=no,downscript=no
```

where:

— -m = memory to assign

— -smp = number of smp cores

— -hda = virtual disk image

This command starts a new virtual machine with two emulated 82540EM devices, backed up with two TAP networking host interfaces, tapvm01 and tapvm02.

```
# ip tuntap show
tapvm01: tap
tapvm02: tap
```

4. Configure your TAP networking interfaces using ip/ifconfig tools.

5. Log in to the guest OS and check that the expected emulated devices exist:

```
# lspci -d 8086:100e
00:04.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev␣
↪03)
00:05.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev␣
↪03)
```

6. Install the DPDK and run testpmd.

9.15.4 Known Limitations of Emulated Devices

The following are known limitations:

1. The Qemu e1000 RX path does not support multiple descriptors/buffers per packet. Therefore, rte_mbuf should be big enough to hold the whole packet. For example, to allow testpmd to receive jumbo frames, use the following:

   testpmd [options] – –mbuf-size=<your-max-packet-size>

2. Qemu e1000 does not validate the checksum of incoming packets.

3. Qemu e1000 only supports one interrupt source, so link and Rx interrupt should be exclusive.

4. Qemu e1000 does not support interrupt auto-clear, application should disable interrupt immediately when woken up.

9.16 ENA Poll Mode Driver

The ENA PMD is a DPDK poll-mode driver for the Amazon Elastic Network Adapter (ENA) family.

9.16.1 Overview

The ENA driver exposes a lightweight management interface with a minimal set of memory mapped registers and an extendable command set through an Admin Queue.

The driver supports a wide range of ENA adapters, is link-speed independent (i.e., the same driver is used for 10GbE, 25GbE, 40GbE, etc.), and it negotiates and supports an extendable feature set.

ENA adapters allow high speed and low overhead Ethernet traffic processing by providing a dedicated Tx/Rx queue pair per CPU core.

The ENA driver supports industry standard TCP/IP offload features such as checksum offload and TCP transmit segmentation offload (TSO).

Receive-side scaling (RSS) is supported for multi-core scaling.

Some of the ENA devices support a working mode called Low-latency Queue (LLQ), which saves several more microseconds.

9.16.2 Management Interface

ENA management interface is exposed by means of:

- Device Registers
- Admin Queue (AQ) and Admin Completion Queue (ACQ)

ENA device memory-mapped PCIe space for registers (MMIO registers) are accessed only during driver initialization and are not involved in further normal device operation.

AQ is used for submitting management commands, and the results/responses are reported asynchronously through ACQ.

ENA introduces a very small set of management commands with room for vendor-specific extensions. Most of the management operations are framed in a generic Get/Set feature command.

The following admin queue commands are supported:

- Create I/O submission queue
- Create I/O completion queue
- Destroy I/O submission queue
- Destroy I/O completion queue
- Get feature
- Set feature
- Get statistics

Refer to `ena_admin_defs.h` for the list of supported Get/Set Feature properties.

9.16.3 Data Path Interface

I/O operations are based on Tx and Rx Submission Queues (Tx SQ and Rx SQ correspondingly). Each SQ has a completion queue (CQ) associated with it.

The SQs and CQs are implemented as descriptor rings in contiguous physical memory.

Refer to `ena_eth_io_defs.h` for the detailed structure of the descriptor

The driver supports multi-queue for both Tx and Rx.

9.16.4 Configuration information

DPDK Configuration Parameters

The following configuration options are available for the ENA PMD:

- **CONFIG_RTE_LIBRTE_ENA_PMD** (default y): Enables or disables inclusion of the ENA PMD driver in the DPDK compilation.
- **CONFIG_RTE_LIBRTE_ENA_DEBUG_RX** (default n): Enables or disables debug logging of RX logic within the ENA PMD driver.
- **CONFIG_RTE_LIBRTE_ENA_DEBUG_TX** (default n): Enables or disables debug logging of TX logic within the ENA PMD driver.
- **CONFIG_RTE_LIBRTE_ENA_COM_DEBUG** (default n): Enables or disables debug logging of low level tx/rx logic in ena_com(base) within the ENA PMD driver.

Runtime Configuration Parameters

- **large_llq_hdr** (default 0)

  Enables or disables usage of large LLQ headers. This option will have effect only if the device also supports large LLQ headers. Otherwise, the default value will be used.

ENA Configuration Parameters

- **Number of Queues**

  This is the requested number of queues upon initialization, however, the actual number of receive and transmit queues to be created will be the minimum between the maximal number supported by the device and number of queues requested.

- **Size of Queues**

  This is the requested size of receive/transmit queues, while the actual size will be the minimum between the requested size and the maximal receive/transmit supported by the device.

9.16.5 Building DPDK

See the *DPDK Getting Started Guide for Linux* for instructions on how to build DPDK.

By default the ENA PMD library will be built into the DPDK library.

For configuring and using UIO and VFIO frameworks, please also refer *the documentation that comes with DPDK suite*.

9.16.6 Supported ENA adapters

Current ENA PMD supports the following ENA adapters including:

- `1d0f:ec20` - ENA VF
- `1d0f:ec21` - ENA VF with LLQ support

9.16.7 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in `System Requirements` section of *the DPDK documentation* or refer to *DPDK Release Notes*.

9.16.8 Supported features

- MTU configuration
- Jumbo frames up to 9K
- IPv4/TCP/UDP checksum offload
- TSO offload
- Multiple receive and transmit queues
- RSS hash
- RSS indirection table configuration
- Low Latency Queue for Tx
- Basic and extended statistics
- LSC event notification
- Watchdog (requires handling of timers in the application)
- Device reset upon failure

9.16.9 Prerequisites

1. Prepare the system as recommended by DPDK suite. This includes environment variables, hugepages configuration, tool-chains and configuration.

2. ENA PMD can operate with `vfio-pci``(*) or ``igb_uio` driver.

   (*) ENAv2 hardware supports Low Latency Queue v2 (LLQv2). This feature reduces the latency of the packets by pushing the header directly through the PCI to the device, before the DMA is even triggered. For proper work kernel PCI driver must support write combining (WC). In mainline version of `igb_uio` (in DPDK repo) it must be enabled by loading module with `wc_activate=1` flag (example below). However, mainline's vfio-pci driver in kernel doesn't have WC support yet (planed to be added). If vfio-pci used user should be either turn off ENAv2 (to avoid performance impact) or recompile vfio-pci driver with patch provided in amzn-github.

3. Insert `vfio-pci` or `igb_uio` kernel module using the command `modprobe vfio-pci` or `modprobe uio; insmod igb_uio.ko wc_activate=1` respectively.

4. For `vfio-pci` users only: Please make sure that `IOMMU` is enabled in your system, or use `vfio` driver in `noiommu` mode:

   ```
   echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode
   ```

   To use `noiommu` mode, the `vfio-pci` must be built with flag `CONFIG_VFIO_NOIOMMU`.

5. Bind the intended ENA device to `vfio-pci` or `igb_uio` module.

At this point the system should be ready to run DPDK applications. Once the application runs to completion, the ENA can be detached from attached module if necessary.

Note about usage on *.metal instances

On AWS, the metal instances are supporting IOMMU for both arm64 and x86_64 hosts.

- **x86_64 (e.g. c5.metal, i3.metal):**
  IOMMU should be disabled by default. In that situation, the `igb_uio` can be used as it is but `vfio-pci` should be working in no-IOMMU mode (please see above).

  When IOMMU is enabled, `igb_uio` cannot be used as it's not supporting this feature, while `vfio-pci` should work without any changes. To enable IOMMU on those hosts, please update `GRUB_CMDLINE_LINUX` in file `/etc/default/grub` with the below extra boot arguments:

  ```
  iommu=1 intel_iommu=on
  ```

  Then, make the changes live by executing as a root:

  ```
  # grub2-mkconfig > /boot/grub2/grub.cfg
  ```

  Finally, reboot should result in IOMMU being enabled.

- **arm64 (a1.metal):**
  IOMMU should be enabled by default. Unfortunately, `vfio-pci` isn't supporting SMMU, which is implementation of IOMMU for arm64 architecture and `igb_uio` isn't supporting IOMMU at all, so to use DPDK with ENA on those hosts, one must disable IOMMU. This can be done by updating `GRUB_CMDLINE_LINUX` in file `/etc/default/grub` with the extra boot argument:

```
iommu.passthrough=1
```

Then, make the changes live by executing as a root:

```
# grub2-mkconfig > /boot/grub2/grub.cfg
```

Finally, reboot should result in IOMMU being disabled. Without IOMMU, `igb_uio` can be used as it is but `vfio-pci` should be working in no-IOMMU mode (please see above).

9.16.10 Usage example

Follow instructions available in the document *compiling and testing a PMD for a NIC* to launch **testpmd** with Amazon ENA devices managed by librte_pmd_ena.

Example output:

```
[...]
EAL: PCI device 0000:00:06.0 on NUMA socket -1
EAL:   Invalid NUMA socket, default to 0
EAL:   probe driver: 1d0f:ec20 net_ena

Interactive-mode selected
testpmd: create a new mbuf pool <mbuf_pool_socket_0>: n=171456, size=2176, socket=0
testpmd: preferred mempool ops selected: ring_mp_mc
Warning! port-topology=paired and odd forward ports number, the last port will pair with␣
↪itself.
Configuring Port 0 (socket 0)
Port 0: 00:00:00:11:00:01
Checking link statuses...

Done
testpmd>
```

9.17 ENETC Poll Mode Driver

The ENETC NIC PMD (**librte_pmd_enetc**) provides poll mode driver support for the inbuilt NIC found in the **NXP LS1028** SoC.

More information can be found at NXP Official Website.

9.17.1 ENETC

This section provides an overview of the NXP ENETC and how it is integrated into the DPDK.

Contents summary

- ENETC overview
- ENETC features
- PCI bus driver
- NIC driver
- Supported ENETC SoCs

- Prerequisites
- Driver compilation and testing

ENETC Overview

ENETC is a PCI Integrated End Point(IEP). IEP implements peripheral devices in an SoC such that software sees them as PCIe device. ENETC is an evolution of BDR(Buffer Descriptor Ring) based networking IPs.

This infrastructure simplifies adding support for IEP and facilitates in following:

- Device discovery and location
- Resource requirement discovery and allocation (e.g. interrupt assignment, device register address)
- Event reporting

ENETC Features

- Link Status
- Packet type information
- Basic stats
- Promiscuous
- Multicast
- Jumbo packets
- Queue Start/Stop
- Deferred Queue Start
- CRC offload

NIC Driver (PMD)

ENETC PMD is traditional DPDK PMD which provides necessary interface between RTE framework and ENETC internal drivers.

- Driver registers the device vendor table in PCI subsystem.
- RTE framework scans the PCI bus for connected devices.
- This scanning will invoke the probe function of ENETC driver.
- The probe function will set the basic device registers and also setups BD rings.
- On packet Rx the respective BD Ring status bit is set which is then used for packet processing.
- Then Tx is done first followed by Rx.

Supported ENETC SoCs

- LS1028

Prerequisites

There are three main pre-requisites for executing ENETC PMD on a ENETC compatible board:

1. **ARM 64 Tool Chain**

   For example, the *aarch64* Linaro Toolchain.

2. **Linux Kernel**

   It can be obtained from NXP's Github hosting.

3. **Rootfile system**

   Any *aarch64* supporting filesystem can be used. For example, Ubuntu 16.04 LTS (Xenial) or 18.04 (Bionic) userland which can be obtained from here.

The following dependencies are not part of DPDK and must be installed separately:

- **NXP Linux LSDK**

  NXP Layerscape software development kit (LSDK) includes support for family of QorIQ® ARM-Architecture-based system on chip (SoC) processors and corresponding boards.

  It includes the Linux board support packages (BSPs) for NXP SoCs, a fully operational tool chain, kernel and board specific modules.

  LSDK and related information can be obtained from: LSDK

Driver compilation and testing

Follow instructions available in the document *compiling and testing a PMD for a NIC* to launch **testpmd**

To compile in performance mode, please set `CONFIG_RTE_CACHE_LINE_SIZE=64`

9.18 ENIC Poll Mode Driver

ENIC PMD is the DPDK poll-mode driver for the Cisco System Inc. VIC Ethernet NICs. These adapters are also referred to as vNICs below. If you are running or would like to run DPDK software applications on Cisco UCS servers using Cisco VIC adapters the following documentation is relevant.

9.18.1 How to obtain ENIC PMD integrated DPDK

ENIC PMD support is integrated into the DPDK suite. dpdk-<version>.tar.gz should be downloaded from https://core.dpdk.org/download/

9.18.2 Configuration information

- **DPDK Configuration Parameters**

  The following configuration options are available for the ENIC PMD:

  - **CONFIG_RTE_LIBRTE_ENIC_PMD** (default y): Enables or disables inclusion of the ENIC PMD driver in the DPDK compilation.

- **vNIC Configuration Parameters**

  - **Number of Queues**

    The maximum number of receive queues (RQs), work queues (WQs) and completion queues (CQs) are configurable on a per vNIC basis through the Cisco UCS Manager (CIMC or UCSM).

    These values should be configured as follows:

    * The number of WQs should be greater or equal to the value of the expected nb_tx_q parameter in the call to rte_eth_dev_configure()

    * The number of RQs configured in the vNIC should be greater or equal to *twice* the value of the expected nb_rx_q parameter in the call to rte_eth_dev_configure(). With the addition of Rx scatter, a pair of RQs on the vnic is needed for each receive queue used by DPDK, even if Rx scatter is not being used. Having a vNIC with only 1 RQ is not a valid configuration, and will fail with an error message.

    * The number of CQs should set so that there is one CQ for each WQ, and one CQ for each pair of RQs.

    For example: If the application requires 3 Rx queues, and 3 Tx queues, the vNIC should be configured to have at least 3 WQs, 6 RQs (3 pairs), and 6 CQs (3 for use by WQs + 3 for use by the 3 pairs of RQs).

  - **Size of Queues**

    Likewise, the number of receive and transmit descriptors are configurable on a per-vNIC basis via the UCS Manager and should be greater than or equal to the nb_rx_desc and nb_tx_desc parameters expected to be used in the calls to rte_eth_rx_queue_setup() and rte_eth_tx_queue_setup() respectively. An application requesting more than the set size will be limited to that size.

    Unless there is a lack of resources due to creating many vNICs, it is recommended that the WQ and RQ sizes be set to the maximum. This gives the application the greatest amount of flexibility in its queue configuration.

    * *Note*: Since the introduction of Rx scatter, for performance reasons, this PMD uses two RQs on the vNIC per receive queue in DPDK. One RQ holds descriptors for the start of a packet, and the second RQ holds the descriptors for the rest of the fragments of a packet. This means that the nb_rx_desc parameter to rte_eth_rx_queue_setup() can be a greater than 4096. The exact amount will depend on the size of the mbufs being used for receives, and the MTU size.

For example: If the mbuf size is 2048, and the MTU is 9000, then receiving a full size packet will take 5 descriptors, 1 from the start-of-packet queue, and 4 from the second queue. Assuming that the RQ size was set to the maximum of 4096, then the application can specify up to 1024 + 4096 as the nb_rx_desc parameter to rte_eth_rx_queue_setup().

– **Interrupts**

At least one interrupt per vNIC interface should be configured in the UCS manager regardless of the number receive/transmit queues. The ENIC PMD uses this interrupt to get information about link status and errors in the fast path.

In addition to the interrupt for link status and errors, when using Rx queue interrupts, increase the number of configured interrupts so that there is at least one interrupt for each Rx queue. For example, if the app uses 3 Rx queues and wants to use per-queue interrupts, configure 4 (3 + 1) interrupts.

– **Receive Side Scaling**

In order to fully utilize RSS in DPDK, enable all RSS related settings in CIMC or UCSM. These include the following items listed under Receive Side Scaling: TCP, IPv4, TCP-IPv4, IPv6, TCP-IPv6, IPv6 Extension, TCP-IPv6 Extension.

9.18.3 SR-IOV mode utilization

UCS blade servers configured with dynamic vNIC connection policies in UCSM are capable of supporting SR-IOV. SR-IOV virtual functions (VFs) are specialized vNICs, distinct from regular Ethernet vNICs. These VFs can be directly assigned to virtual machines (VMs) as 'passthrough' devices.

In UCS, SR-IOV VFs require the use of the Cisco Virtual Machine Fabric Extender (VM-FEX), which gives the VM a dedicated interface on the Fabric Interconnect (FI). Layer 2 switching is done at the FI. This may eliminate the requirement for software switching on the host to route intra-host VM traffic.

Please refer to Creating a Dynamic vNIC Connection Policy for information on configuring SR-IOV adapter policies and port profiles using UCSM.

Once the policies are in place and the host OS is rebooted, VFs should be visible on the host, E.g.:

```
# lspci | grep Cisco | grep Ethernet
0d:00.0 Ethernet controller: Cisco Systems Inc VIC Ethernet NIC (rev a2)
0d:00.1 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.2 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.3 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.4 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.5 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.6 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.7 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
```

Enable Intel IOMMU on the host and install KVM and libvirt, and reboot again as required. Then, using libvirt, create a VM instance with an assigned device. Below is an example `interface` block (part of the domain configuration XML) that adds the host VF 0d:00:01 to the VM. `profileid='pp-vlan-25'` indicates the port profile that has been configured in UCSM.

```
<interface type='hostdev' managed='yes'>
  <mac address='52:54:00:ac:ff:b6'/>
  <driver name='vfio'/>
  <source>
    <address type='pci' domain='0x0000' bus='0x0d' slot='0x00' function='0x1'/>
```

(continues on next page)

(continued from previous page)

```
  </source>
  <virtualport type='802.1Qbh'>
    <parameters profileid='pp-vlan-25'/>
  </virtualport>
</interface>
```

Alternatively, the configuration can be done in a separate file using the `network` keyword. These methods are described in the libvirt documentation for Network XML format.

When the VM instance is started, libvirt will bind the host VF to vfio, complete provisioning on the FI and bring up the link.

Note: It is not possible to use a VF directly from the host because it is not fully provisioned until libvirt brings up the VM that it is assigned to.

In the VM instance, the VF will now be visible. E.g., here the VF 00:04.0 is seen on the VM instance and should be available for binding to a DPDK.

```
# lspci | grep Ether
00:04.0 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
```

Follow the normal DPDK install procedure, binding the VF to either `igb_uio` or `vfio` in non-IOMMU mode.

In the VM, the kernel enic driver may be automatically bound to the VF during boot. Unbinding it currently hangs due to a known issue with the driver. To work around the issue, blacklist the enic module as follows. Please see *Limitations* for limitations in the use of SR-IOV.

```
# cat /etc/modprobe.d/enic.conf
blacklist enic

# dracut --force
```

Note: Passthrough does not require SR-IOV. If VM-FEX is not desired, the user may create as many regular vNICs as necessary and assign them to VMs as passthrough devices. Since these vNICs are not SR-IOV VFs, using them as passthrough devices do not require libvirt, port profiles, and VM-FEX.

9.18.4 Generic Flow API support

Generic Flow API (also called "rte_flow" API) is supported. More advanced capabilities are available when "Advanced Filtering" is enabled on the adapter. Advanced filtering was added to 1300 series VIC firmware starting with version 2.0.13 for C-series UCS servers and version 3.1.2 for UCSM managed blade servers. Advanced filtering is available on 1400 series adapters and beyond. To enable advanced filtering, the 'Advanced filter' radio button should be selected via CIMC or UCSM followed by a reboot of the server.

- **1200 series VICs**

  5-tuple exact flow support for 1200 series adapters. This allows:

  - Attributes: ingress

  - Items: ipv4, ipv6, udp, tcp (must exactly match src/dst IP addresses and ports and all must be specified)
  - Actions: queue and void
  - Selectors: 'is'

- **1300 and later series VICS with advanced filters disabled**

  With advanced filters disabled, an IPv4 or IPv6 item must be specified in the pattern.

  - Attributes: ingress
  - Items: eth, vlan, ipv4, ipv6, udp, tcp, vxlan, inner eth, vlan, ipv4, ipv6, udp, tcp
  - Actions: queue and void
  - Selectors: 'is', 'spec' and 'mask'. 'last' is not supported
  - In total, up to 64 bytes of mask is allowed across all headers

- **1300 and later series VICS with advanced filters enabled**

  - Attributes: ingress
  - Items: eth, vlan, ipv4, ipv6, udp, tcp, vxlan, raw, inner eth, vlan, ipv4, ipv6, udp, tcp
  - Actions: queue, mark, drop, flag, rss, passthru, and void
  - Selectors: 'is', 'spec' and 'mask'. 'last' is not supported
  - In total, up to 64 bytes of mask is allowed across all headers

- **1400 and later series VICs with Flow Manager API enabled**

  - Attributes: ingress, egress
  - Items: eth, vlan, ipv4, ipv6, sctp, udp, tcp, vxlan, raw, inner eth, vlan, ipv4, ipv6, sctp, udp, tcp
  - Ingress Actions: count, drop, flag, jump, mark, port_id, passthru, queue, rss, vxlan_decap, vxlan_encap, and void
  - Egress Actions: count, drop, jump, passthru, vxlan_encap, and void
  - Selectors: 'is', 'spec' and 'mask'. 'last' is not supported
  - In total, up to 64 bytes of mask is allowed across all headers

The VIC performs packet matching after applying VLAN strip. If VLAN stripping is enabled, EtherType in the ETH item corresponds to the stripped VLAN header's EtherType. Stripping does not affect the VLAN item. TCI and EtherType in the VLAN item are matched against those in the (stripped) VLAN header whether stripping is enabled or disabled.

More features may be added in future firmware and new versions of the VIC. Please refer to the release notes.

9.18.5 Overlay Offload

Recent hardware models support overlay offload. When enabled, the NIC performs the following operations for VXLAN, NVGRE, and GENEVE packets. In all cases, inner and outer packets can be IPv4 or IPv6.

- TSO for VXLAN and GENEVE packets.

  Hardware supports NVGRE TSO, but DPDK currently has no NVGRE offload flags.

- Tx checksum offloads.

  The NIC fills in IPv4/UDP/TCP checksums for both inner and outer packets.

- Rx checksum offloads.

  The NIC validates IPv4/UDP/TCP checksums of both inner and outer packets. Good checksum flags (e.g. `PKT_RX_L4_CKSUM_GOOD`) indicate that the inner packet has the correct checksum, and if applicable, the outer packet also has the correct checksum. Bad checksum flags (e.g. `PKT_RX_L4_CKSUM_BAD`) indicate that the inner and/or outer packets have invalid checksum values.

- Inner Rx packet type classification

  PMD sets inner L3/L4 packet types (e.g. `RTE_PTYPE_INNER_L4_TCP`), and `RTE_PTYPE_TUNNEL_GRENAT` to indicate that the packet is tunneled. PMD does not set L3/L4 packet types for outer packets.

- Inner RSS

  RSS hash calculation, therefore queue selection, is done on inner packets.

In order to enable overlay offload, the 'Enable VXLAN' box should be checked via CIMC or UCSM followed by a reboot of the server. When PMD successfully enables overlay offload, it prints the following message on the console.

```
Overlay offload is enabled
```

By default, PMD enables overlay offload if hardware supports it. To disable it, set `devargs` parameter `disable-overlay=1`. For example:

```
-w 12:00.0,disable-overlay=1
```

By default, the NIC uses 4789 as the VXLAN port. The user may change it through `rte_eth_dev_udp_tunnel_port_{add,delete}`. However, as the current NIC has a single VXLAN port number, the user cannot configure multiple port numbers.

Geneve headers with non-zero options are not supported by default. To use Geneve with options, update the VIC firmware to the latest version and then set `devargs` parameter `geneve-opt=1`. When Geneve with options is enabled, flow API cannot be used as the features are currently mutually exclusive. When this feature is successfully enabled, PMD prints the following message.

```
Geneve with options is enabled
```

9.18.6 Ingress VLAN Rewrite

VIC adapters can tag, untag, or modify the VLAN headers of ingress packets. The ingress VLAN rewrite mode controls this behavior. By default, it is set to pass-through, where the NIC does not modify the VLAN header in any way so that the application can see the original header. This mode is sufficient for many applications, but may not be suitable for others. Such applications may change the mode by setting `devargs` parameter `ig-vlan-rewrite` to one of the following.

- `pass`: Pass-through mode. The NIC does not modify the VLAN header. This is the default mode.
- `priority`: Priority-tag default VLAN mode. If the ingress packet is tagged with the default VLAN, the NIC replaces its VLAN header with the priority tag (VLAN ID 0).
- `trunk`: Default trunk mode. The NIC tags untagged ingress packets with the default VLAN. Tagged ingress packets are not modified. To the application, every packet appears as tagged.
- `untag`: Untag default VLAN mode. If the ingress packet is tagged with the default VLAN, the NIC removes or untags its VLAN header so that the application sees an untagged packet. As a result, the default VLAN becomes *untagged*. This mode can be useful for applications such as OVS-DPDK performance benchmarks that utilize only the default VLAN and want to see only untagged packets.

9.18.7 Vectorized Rx Handler

ENIC PMD includes a version of the receive handler that is vectorized using AVX2 SIMD instructions. It is meant for bulk, throughput oriented workloads where reducing cycles/packet in PMD is a priority. In order to use the vectorized handler, take the following steps.

- Use a recent version of gcc, icc, or clang and build 64-bit DPDK. If the compiler is known to support AVX2, DPDK build system automatically compiles the vectorized handler. Otherwise, the handler is not available.
- Set `devargs` parameter `enable-avx2-rx=1` to explicitly request that PMD consider the vectorized handler when selecting the receive handler. For example:

```
-w 12:00.0,enable-avx2-rx=1
```

  As the current implementation is intended for field trials, by default, the vectorized handler is not considered (`enable-avx2-rx=0`).

- Run on a UCS M4 or later server with CPUs that support AVX2.

PMD selects the vectorized handler when the handler is compiled into the driver, the user requests its use via `enable-avx2-rx=1`, CPU supports AVX2, and scatter Rx is not used. To verify that the vectorized handler is selected, enable debug logging (`--log-level=pmd,debug`) and check the following message.

```
enic_use_vector_rx_handler use the non-scatter avx2 Rx handler
```

9.18.8 Limitations

- **VLAN 0 Priority Tagging**

  If a vNIC is configured in TRUNK mode by the UCS manager, the adapter will priority tag egress packets according to 802.1Q if they were not already VLAN tagged by software. If the adapter is connected to a properly configured switch, there will be no unexpected behavior.

  In test setups where an Ethernet port of a Cisco adapter in TRUNK mode is connected point-to-point to another adapter port or connected though a router instead of a switch, all ingress packets will be VLAN tagged. Programs such as l3fwd may not account for VLAN tags in packets and may misbehave. One solution is to enable VLAN stripping on ingress so the VLAN tag is removed from the packet and put into the mbuf->vlan_tci field. Here is an example of how to accomplish this:

```
vlan_offload = rte_eth_dev_get_vlan_offload(port);
vlan_offload |= ETH_VLAN_STRIP_OFFLOAD;
rte_eth_dev_set_vlan_offload(port, vlan_offload);
```

Another alternative is modify the adapter's ingress VLAN rewrite mode so that packets with the default VLAN tag are stripped by the adapter and presented to DPDK as untagged packets. In this case mbuf->vlan_tci and the PKT_RX_VLAN and PKT_RX_VLAN_STRIPPED mbuf flags would not be set. This mode is enabled with the `devargs` parameter `ig-vlan-rewrite=untag`. For example:

```
-w 12:00.0,ig-vlan-rewrite=untag
```

- **SR-IOV**
  - KVM hypervisor support only. VMware has not been tested.
  - Requires VM-FEX, and so is only available on UCS managed servers connected to Fabric Interconnects. It is not on standalone C-Series servers.
  - VF devices are not usable directly from the host. They can only be used as assigned devices on VM instances.
  - Currently, unbind of the ENIC kernel mode driver 'enic.ko' on the VM instance may hang. As a workaround, enic.ko should be blacklisted or removed from the boot process.
  - pci_generic cannot be used as the uio module in the VM. igb_uio or vfio in non-IOMMU mode can be used.
  - The number of RQs in UCSM dynamic vNIC configurations must be at least 2.
  - The number of SR-IOV devices is limited to 256. Components on target system might limit this number to fewer than 256.
- **Flow API**
  - The number of filters that can be specified with the Generic Flow API is dependent on how many header fields are being masked. Use 'flow create' in a loop to determine how many filters your VIC will support (not more than 1000 for 1300 series VICs). Filters are checked for matching in the order they were added. Since there currently is no grouping or priority support, 'catch-all' filters should be added last.
  - The supported range of IDs for the 'MARK' action is 0 - 0xFFFD.

  - RSS and PASSTHRU actions only support "receive normally". They are limited to supporting MARK + RSS and PASSTHRU + MARK to allow the application to mark packets and then receive them normally. These require 1400 series VIC adapters and latest firmware.
  - RAW items are limited to matching UDP tunnel headers like VXLAN.
  - For 1400 VICs, all flows using the RSS action on a port use same hash configuration. The RETA is ignored. The queues used in the RSS group must be sequential. There is a performance hit if the number of queues is not a power of 2. Only level 0 (outer header) RSS is allowed.

- **Statistics**
  - `rx_good_bytes` (ibytes) always includes VLAN header (4B) and CRC bytes (4B). This behavior applies to 1300 and older series VIC adapters. 1400 series VICs do not count CRC bytes, and count VLAN header only when VLAN stripping is disabled.
  - When the NIC drops a packet because the Rx queue has no free buffers, `rx_good_bytes` still increments by 4B if the packet is not VLAN tagged or VLAN stripping is disabled, or by 8B if the packet is VLAN tagged and stripping is enabled. This behavior applies to 1300 and older series VIC adapters. 1400 series VICs do not increment this byte counter when packets are dropped.

- **RSS Hashing**
  - Hardware enables and disables UDP and TCP RSS hashing together. The driver cannot control UDP and TCP hashing individually.

9.18.9 How to build the suite

The build instructions for the DPDK suite should be followed. By default the ENIC PMD library will be built into the DPDK library.

Refer to the document *compiling and testing a PMD for a NIC* for details.

For configuring and using UIO and VFIO frameworks, please refer to the documentation that comes with DPDK suite.

9.18.10 Supported Cisco VIC adapters

ENIC PMD supports all recent generations of Cisco VIC adapters including:

- VIC 1200 series
- VIC 1300 series
- VIC 1400 series

9.18.11 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in Dependencies section of DPDK documentation.

9.18.12 Supported features

- Unicast, multicast and broadcast transmission and reception
- Receive queue polling
- Port Hardware Statistics
- Hardware VLAN acceleration
- IP checksum offload
- Receive side VLAN stripping
- Multiple receive and transmit queues
- Promiscuous mode
- Setting RX VLAN (supported via UCSM/CIMC only)
- VLAN filtering (supported via UCSM/CIMC only)
- Execution of application by unprivileged system users
- IPV4, IPV6 and TCP RSS hashing
- UDP RSS hashing (1400 series and later adapters)
- Scattered Rx
- MTU update
- SR-IOV on UCS managed servers connected to Fabric Interconnects
- Flow API
- Overlay offload
  - Rx/Tx checksum offloads for VXLAN, NVGRE, GENEVE
  - TSO for VXLAN and GENEVE packets
  - Inner RSS

9.18.13 Known bugs and unsupported features in this release

- Signature or flex byte based flow direction
- Drop feature of flow direction
- VLAN based flow direction
- Non-IPV4 flow direction
- Setting of extended VLAN
- MTU update only works if Scattered Rx mode is disabled

- Maximum receive packet length is ignored if Scattered Rx mode is used

9.18.14 Prerequisites

- Prepare the system as recommended by DPDK suite. This includes environment variables, hugepages configuration, tool-chains and configuration.
- Insert vfio-pci kernel module using the command 'modprobe vfio-pci' if the user wants to use VFIO framework.
- Insert uio kernel module using the command 'modprobe uio' if the user wants to use UIO framework.
- DPDK suite should be configured based on the user's decision to use VFIO or UIO framework.
- If the vNIC device(s) to be used is bound to the kernel mode Ethernet driver use 'ip' to bring the interface down. The dpdk-devbind.py tool can then be used to unbind the device's bus id from the ENIC kernel mode driver.
- Bind the intended vNIC to vfio-pci in case the user wants ENIC PMD to use VFIO framework using dpdk-devbind.py.
- Bind the intended vNIC to igb_uio in case the user wants ENIC PMD to use UIO framework using dpdk-devbind.py.

At this point the system should be ready to run DPDK applications. Once the application runs to completion, the vNIC can be detached from vfio-pci or igb_uio if necessary.

Root privilege is required to bind and unbind vNICs to/from VFIO/UIO. VFIO framework helps an unprivileged user to run the applications. For an unprivileged user to run the applications on DPDK and ENIC PMD, it may be necessary to increase the maximum locked memory of the user. The following command could be used to do this.

```
sudo sh -c "ulimit -l <value in Kilo Bytes>"
```

The value depends on the memory configuration of the application, DPDK and PMD. Typically, the limit has to be raised to higher than 2GB. e.g., 2621440

The compilation of any unused drivers can be disabled using the configuration file in config/ directory (e.g., config/common_linux). This would help in bringing down the time taken for building the libraries and the initialization time of the application.

9.18.15 Additional Reference

- https://www.cisco.com/c/en/us/products/servers-unified-computing/index.html
- https://www.cisco.com/c/en/us/products/interfaces-modules/unified-computing-system-adapters/index.html

9.18.16 Contact Information

Any questions or bugs should be reported to DPDK community and to the ENIC PMD maintainers:

- John Daley <johndale@cisco.com>
- Hyong Youb Kim <hyonkim@cisco.com>

9.19 FM10K Poll Mode Driver

The FM10K poll mode driver library provides support for the Intel FM10000 (FM10K) family of 40GbE/100GbE adapters.

9.19.1 FTAG Based Forwarding of FM10K

FTAG Based Forwarding is a unique feature of FM10K. The FM10K family of NICs support the addition of a Fabric Tag (FTAG) to carry special information. The FTAG is placed at the beginning of the frame, it contains information such as where the packet comes from and goes, and the vlan tag. In FTAG based forwarding mode, the switch logic forwards packets according to glort (global resource tag) information, rather than the mac and vlan table. Currently this feature works only on PF.

To enable this feature, the user should pass a devargs parameter to the eal like "-w 84:00.0,enable_ftag=1", and the application should make sure an appropriate FTAG is inserted for every frame on TX side.

9.19.2 Vector PMD for FM10K

Vector PMD (vPMD) uses Intel® SIMD instructions to optimize packet I/O. It improves load/store bandwidth efficiency of L1 data cache by using a wider SSE/AVX ''register (1)". The wider register gives space to hold multiple packet buffers so as to save on the number of instructions when bulk processing packets.

There is no change to the PMD API. The RX/TX handlers are the only two entries for vPMD packet I/O. They are transparently registered at runtime RX/TX execution if all required conditions are met.

1. To date, only an SSE version of FM10K vPMD is available. To ensure that vPMD is in the binary code, set `CONFIG_RTE_LIBRTE_FM10K_INC_VECTOR=y` in the configure file.

Some constraints apply as pre-conditions for specific optimizations on bulk packet transfers. The following sections explain RX and TX constraints in the vPMD.

RX Constraints

Prerequisites and Pre-conditions

For Vector RX it is assumed that the number of descriptor rings will be a power of 2. With this pre-condition, the ring pointer can easily scroll back to the head after hitting the tail without a conditional check. In addition Vector RX can use this assumption to do a bit mask using `ring_size - 1`.

Features not Supported by Vector RX PMD

Some features are not supported when trying to increase the throughput in vPMD. They are:

- IEEE1588
- Flow director
- Header split
- RX checksum offload

Other features are supported using optional MACRO configuration. They include:

- HW VLAN strip
- L3/L4 packet type

To enable via `RX_OLFLAGS` use `RTE_LIBRTE_FM10K_RX_OLFLAGS_ENABLE=y`.

To guarantee the constraint, the following capabilities in `dev_conf.rxmode.offloads` will be checked:

- `DEV_RX_OFFLOAD_VLAN_EXTEND`
- `DEV_RX_OFFLOAD_CHECKSUM`
- `DEV_RX_OFFLOAD_HEADER_SPLIT`
- `fdir_conf->mode`

RX Burst Size

As vPMD is focused on high throughput, it processes 4 packets at a time. So it assumes that the RX burst should be greater than 4 packets per burst. It returns zero if using `nb_pkt` < 4 in the receive handler. If `nb_pkt` is not a multiple of 4, a floor alignment will be applied.

TX Constraint

Features not Supported by TX Vector PMD

TX vPMD only works when offloads is set to 0

This means that it does not support any TX offload.

9.19.3 Limitations

Switch manager

The Intel FM10000 family of NICs integrate a hardware switch and multiple host interfaces. The FM10000 PMD driver only manages host interfaces. For the switch component another switch driver has to be loaded prior to the FM10000 PMD driver. The switch driver can be acquired from Intel support. Only Testpoint is validated with DPDK, the latest version that has been validated with DPDK is 4.1.6.

Support for Switch Restart

For FM10000 multi host based design a DPDK app running in the VM or host needs to be aware of the switch's state since it may undergo a quit-restart. When the switch goes down the DPDK app will receive a LSC event indicating link status down, and the app should stop the worker threads that are polling on the Rx/Tx queues. When switch comes up, a LSC event indicating `LINK_UP` is sent to the app, which can then restart the FM10000 port to resume network processing.

CRC stripping

The FM10000 family of NICs strip the CRC for every packets coming into the host interface. So, keeping CRC is not supported.

Maximum packet length

The FM10000 family of NICS support a maximum of a 15K jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 15364, frames up to 15364 bytes can still reach the host interface.

Statistic Polling Frequency

The FM10000 NICs expose a set of statistics via the PCI BARs. These statistics are read from the hardware registers when `rte_eth_stats_get()` or `rte_eth_xstats_get()` is called. The packet counting registers are 32 bits while the byte counting registers are 48 bits. As a result, the statistics must be polled regularly in order to ensure the consistency of the returned reads.

Given the PCIe Gen3 x8, about 50Gbps of traffic can occur. With 64 byte packets this gives almost 100 million packets/second, causing 32 bit integer overflow after approx 40 seconds. To ensure these overflows are detected and accounted for in the statistics, it is necessary to read statistic regularly. It is suggested to read stats every 20 seconds, which will ensure the statistics are accurate.

Interrupt mode

The FM10000 family of NICS need one separate interrupt for mailbox. So only drivers which support multiple interrupt vectors e.g. vfio-pci can work for fm10k interrupt mode.

9.20 HINIC Poll Mode Driver

The hinic PMD (librte_pmd_hinic) provides poll mode driver support for 25Gbps Huawei Intelligent PCIE Network Adapters based on the Huawei Ethernet Controller Hi1822.

9.20.1 Features

- Multi arch support: x86_64, ARMv8.
- Multiple queues for TX and RX
- Receiver Side Scaling (RSS)
- MAC/VLAN filtering
- Checksum offload
- TSO offload
- Promiscuous mode
- Port hardware statistics
- Link state information
- Link flow control
- Scattered and gather for TX and RX
- SR-IOV - Partially supported at this point, VFIO only
- VLAN filter and VLAN offload
- Allmulticast mode
- MTU update
- Unicast MAC filter
- Multicast MAC filter
- Flow API
- Set Link down or up
- FW version
- LRO

9.20.2 Prerequisites

- Learning about Huawei Hi1822 IN200 Series Intelligent NICs using https://e.huawei.com/en/products/cloud-computing-dc/servers/pcie-ssd/in-card.
- Getting the latest product documents and software supports using https://support.huawei.com/enterprise/en/intelligent-accelerator-components/in500-solution-pid-23507369.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

9.20.3 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_HINIC_PMD` (default `y`)

9.20.4 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

9.20.5 Limitations or Known issues

Build with ICC is not supported yet. X86-32, Power8, ARMv7 and BSD are not supported yet.

9.21 HNS3 Poll Mode Driver

The hns3 PMD (librte_pmd_hns3) provides poll mode driver support for the inbuilt Hisilicon Network Subsystem(HNS) network engine found in the Hisilicon Kunpeng 920 SoC.

9.21.1 Features

Features of the HNS3 PMD are:

- Multiple queues for TX and RX
- Receive Side Scaling (RSS)
- Packet type information
- Checksum offload
- TSO offload
- Promiscuous mode
- Multicast mode
- Port hardware statistics
- Jumbo frames
- Link state information
- Interrupt mode for RX
- VLAN stripping
- NUMA support

9.21.2 Prerequisites

- Get the information about Kunpeng920 chip using http://www.hisilicon.com/en/Products/ProductList/Kunpeng.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

9.21.3 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_HNS3_PMD` (default `y`)

9.21.4 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

9.21.5 Limitations or Known issues

Currently, we only support VF device is bound to vfio_pci or igb_uio and then driven by DPDK driver when PF is driven by kernel mode hns3 ethdev driver, VF is not supported when PF is driven by DPDK driver.

Build with ICC is not supported yet. X86-32, Power8, ARMv7 and BSD are not supported yet.

9.22 I40E Poll Mode Driver

The i40e PMD (librte_pmd_i40e) provides poll mode driver support for 10/25/40 Gbps Intel® Ethernet 700 Series Network Adapters based on the Intel Ethernet Controller X710/XL710/XXV710 and Intel Ethernet Connection X722 (only support part of features).

9.22.1 Features

Features of the i40e PMD are:

- Multiple queues for TX and RX
- Receiver Side Scaling (RSS)
- MAC/VLAN filtering
- Packet type information
- Flow director
- Cloud filter
- Checksum offload
- VLAN/QinQ stripping and inserting

- TSO offload
- Promiscuous mode
- Multicast mode
- Port hardware statistics
- Jumbo frames
- Link state information
- Link flow control
- Mirror on port, VLAN and VSI
- Interrupt mode for RX
- Scattered and gather for TX and RX
- Vector Poll mode driver
- DCB
- VMDQ
- SR-IOV VF
- Hot plug
- IEEE1588/802.1AS timestamping
- VF Daemon (VFD) - EXPERIMENTAL
- Dynamic Device Personalization (DDP)
- Queue region configuration
- Virtual Function Port Representors
- Malicious Device Drive event catch and notify
- Generic flow API

9.22.2 Prerequisites

- Identifying your adapter using Intel Support and get the latest NVM/FW images.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.
- To get better performance on Intel platforms, please follow the "How to get best performance with NICs on Intel platforms" section of the *Getting Started Guide for Linux*.
- Upgrade the NVM/FW version following the Intel® Ethernet NVM Update Tool Quick Usage Guide for Linux and Intel® Ethernet NVM Update Tool: Quick Usage Guide for EFI if needed.
- For information about supported media, please refer to this document: Intel® Ethernet Controller X710/XXV710/XL710 Feature Support Matrix.

Note:

- Some adapters based on the Intel(R) Ethernet Controller 700 Series only support Intel Ethernet Optics modules. On these adapters, other modules are not supported and will not function.
- For connections based on Intel(R) Ethernet Controller 700 Series, support is dependent on your system board. Please see your vendor for details.
- In all cases Intel recommends using Intel Ethernet Optics; other modules may function but are not validated by Intel. Contact Intel for supported media types.

9.22.3 Recommended Matching List

It is highly recommended to upgrade the i40e kernel driver and firmware to avoid the compatibility issues with i40e PMD. Here is the suggested matching list which has been tested and verified. The detailed information can refer to chapter Tested Platforms/Tested NICs in release notes.

For X710/XL710/XXV710,

DPDK version	Kernel driver version	Firmware version
20.05	2.11.27	7.30
20.02	2.10.19	7.20
19.11	2.9.21	7.00
19.08	2.8.43	7.00
19.05	2.7.29	6.80
19.02	2.7.26	6.80
18.11	2.4.6	6.01
18.08	2.4.6	6.01
18.05	2.4.6	6.01
18.02	2.4.3	6.01
17.11	2.1.26	6.01
17.08	2.0.19	6.01
17.05	1.5.23	5.05
17.02	1.5.23	5.05
16.11	1.5.23	5.05
16.07	1.4.25	5.04
16.04	1.4.25	5.02

For X722,

DPDK version	Kernel driver version	Firmware version
20.05	2.11.27	4.11
20.02	2.10.19	4.11
19.11	2.9.21	4.10
19.08	2.9.21	4.10
19.05	2.7.29	3.33
19.02	2.7.26	3.33
18.11	2.4.6	3.33

9.22.4 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_I40E_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_i40e` driver.

- `CONFIG_RTE_LIBRTE_I40E_DEBUG_*` (default `n`)

  Toggle display of generic debugging messages.

- `CONFIG_RTE_LIBRTE_I40E_RX_ALLOW_BULK_ALLOC` (default `y`)

  Toggle bulk allocation for RX.

- `CONFIG_RTE_LIBRTE_I40E_INC_VECTOR` (default `n`)

  Toggle the use of Vector PMD instead of normal RX/TX path. To enable vPMD for RX, bulk allocation for Rx must be allowed.

- `CONFIG_RTE_LIBRTE_I40E_16BYTE_RX_DESC` (default `n`)

  Toggle to use a 16-byte RX descriptor, by default the RX descriptor is 32 byte.

- `CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_PF` (default `64`)

  Number of queues reserved for PF.

- `CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_VM` (default `4`)

  Number of queues reserved for each VMDQ Pool.

Runtime Config Options

- `Reserved number of Queues per VF` (default `4`)

  The number of reserved queue per VF is determined by its host PF. If the PCI address of an i40e PF is aaaa:bb.cc, the number of reserved queues per VF can be configured with EAL parameter like -w aaaa:bb.cc,queue-num-per-vf=n. The value n can be 1, 2, 4, 8 or 16. If no such parameter is configured, the number of reserved queues per VF is 4 by default. If VF request more than reserved queues per VF, PF will able to allocate max to 16 queues after a VF reset.

- `Support multiple driver` (default `disable`)

  There was a multiple driver support issue during use of 700 series Ethernet Adapter with both Linux kernel and DPDK PMD. To fix this issue, `devargs` parameter `support-multi-driver` is introduced, for example:

  ```
  -w 84:00.0,support-multi-driver=1
  ```

  With the above configuration, DPDK PMD will not change global registers, and will switch PF interrupt from IntN to Int0 to avoid interrupt conflict between DPDK and Linux Kernel.

- `Support VF Port Representor` (default `not enabled`)

The i40e PF PMD supports the creation of VF port representors for the control and monitoring of i40e virtual function devices. Each port representor corresponds to a single virtual function of that device. Using the `devargs` option `representor` the user can specify which virtual functions to create port representors for on initialization of the PF PMD by passing the VF IDs of the VFs which are required.:

```
-w DBDF,representor=[0,1,4]
```

Currently hot-plugging of representor ports is not supported so all required representors must be specified on the creation of the PF.

- `Use latest supported vector` (default `disable`)

  Latest supported vector path may not always get the best perf so vector path was recommended to use only on later platform. But users may want the latest vector path since it can get better perf in some real work loading cases. So `devargs` param `use-latest-supported-vec` is introduced, for example:

  ```
  -w 84:00.0,use-latest-supported-vec=1
  ```

- `Enable validation for VF message` (default `not enabled`)

  The PF counts messages from each VF. If in any period of seconds the message statistic from a VF exceeds maximal limitation, the PF will ignore any new message from that VF for some seconds. Format – "maximal-message@period-seconds:ignore-seconds" For example:

  ```
  -w 84:00.0,vf_msg_cfg=80@120:180
  ```

Vector RX Pre-conditions

For Vector RX it is assumed that the number of descriptor rings will be a power of 2. With this pre-condition, the ring pointer can easily scroll back to the head after hitting the tail without a conditional check. In addition Vector RX can use this assumption to do a bit mask using `ring_size - 1`.

9.22.5 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

9.22.6 SR-IOV: Prerequisites and sample Application Notes

1. Load the kernel module:

   ```
   modprobe i40e
   ```

   Check the output in dmesg:

   ```
   i40e 0000:83:00.1 ens802f0: renamed from eth0
   ```

2. Bring up the PF ports:

   ```
   ifconfig ens802f0 up
   ```

3. Create VF device(s):

   Echo the number of VFs to be created into the `sriov_numvfs` sysfs entry of the parent PF.

   Example:

```
echo 2 > /sys/devices/pci0000:00/0000:00:03.0/0000:81:00.0/sriov_numvfs
```

4. Assign VF MAC address:

   Assign MAC address to the VF using iproute2 utility. The syntax is:

```
ip link set <PF netdev id> vf <VF id> mac <macaddr>
```

   Example:

```
ip link set ens802f0 vf 0 mac a0:b0:c0:d0:e0:f0
```

5. Assign VF to VM, and bring up the VM. Please see the documentation for the *I40E/IXGBE/IGB Virtual Function Driver*.

6. Running testpmd:

   Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

   Example output:

```
...
EAL: PCI device 0000:83:00.0 on NUMA socket 1
EAL: probe driver: 8086:1572 rte_i40e_pmd
EAL: PCI memory mapped at 0x7f7f80000000
EAL: PCI memory mapped at 0x7f7f80800000
PMD: eth_i40e_dev_init(): FW 5.0 API 1.5 NVM 05.00.02 eetrack 8000208a
Interactive-mode selected
Configuring Port 0 (socket 0)
...

PMD: i40e_dev_rx_queue_setup(): Rx Burst Bulk Alloc Preconditions are
satisfied.Rx Burst Bulk Alloc function will be used on port=0, queue=0.

...
Port 0: 68:05:CA:26:85:84
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done

testpmd>
```

9.22.7 Sample Application Notes

Vlan filter

Vlan filter only works when Promiscuous mode is off.

To start `testpmd`, and add vlan 10 to port 0:

```
./app/testpmd -l 0-15 -n 4 -- -i --forward-mode=mac
...

testpmd> set promisc 0 off
testpmd> rx_vlan add 10 0
```

Flow Director

The Flow Director works in receive mode to identify specific flows or sets of flows and route them to specific queues. The Flow Director filters can match the different fields for different type of packet: flow type, specific input set per flow type and the flexible payload.

The default input set of each flow type is:

```
ipv4-other : src_ip_address, dst_ip_address
ipv4-frag  : src_ip_address, dst_ip_address
ipv4-tcp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv4-udp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv4-sctp  : src_ip_address, dst_ip_address, src_port, dst_port,
             verification_tag
ipv6-other : src_ip_address, dst_ip_address
ipv6-frag  : src_ip_address, dst_ip_address
ipv6-tcp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv6-udp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv6-sctp  : src_ip_address, dst_ip_address, src_port, dst_port,
             verification_tag
l2_payload : ether_type
```

The flex payload is selected from offset 0 to 15 of packet's payload by default, while it is masked out from matching.

Start `testpmd` with `--disable-rss` and `--pkt-filter-mode=perfect`:

```
./app/testpmd -l 0-15 -n 4 -- -i --disable-rss --pkt-filter-mode=perfect \
              --rxq=8 --txq=8 --nb-cores=8 --nb-ports=1
```

Add a rule to direct `ipv4-udp` packet whose `dst_ip=2.2.2.5, src_ip=2.2.2.3, src_port=32, dst_port=32` to queue 1:

```
testpmd> flow_director_filter 0 mode IP add flow ipv4-udp  \
         src 2.2.2.3 32 dst 2.2.2.5 32 vlan 0 flexbytes () \
         fwd pf queue 1 fd_id 1
```

Check the flow director status:

```
testpmd> show port fdir 0

######################## FDIR infos for port 0      ####################
  MODE:   PERFECT
  SUPPORTED FLOW TYPE:  ipv4-frag ipv4-tcp ipv4-udp ipv4-sctp ipv4-other
                        ipv6-frag ipv6-tcp ipv6-udp ipv6-sctp ipv6-other
                        l2_payload
  FLEX PAYLOAD INFO:
  max_len:       16          payload_limit: 480
  payload_unit:  2           payload_seg:   3
  bitmask_unit:  2           bitmask_num:   2
  MASK:
    vlan_tci: 0x0000,
```

(continues on next page)

(continued from previous page)

```
   src_ipv4: 0x00000000,
   dst_ipv4: 0x00000000,
   src_port: 0x0000,
   dst_port: 0x0000
   src_ipv6: 0x00000000,0x00000000,0x00000000,0x00000000,
   dst_ipv6: 0x00000000,0x00000000,0x00000000,0x00000000
 FLEX PAYLOAD SRC OFFSET:
   L2_PAYLOAD:    0      1      2      3      4      5      6  ...
   L3_PAYLOAD:    0      1      2      3      4      5      6  ...
   L4_PAYLOAD:    0      1      2      3      4      5      6  ...
 FLEX MASK CFG:
   ipv4-udp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   ipv4-tcp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   ipv4-sctp:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   ipv4-other:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   ipv4-frag:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   ipv6-udp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   ipv6-tcp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   ipv6-sctp:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   ipv6-other:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   ipv6-frag:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   l2_payload:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 guarant_count: 1           best_count:    0
 guarant_space: 512         best_space:    7168
 collision:     0           free:          0
 maxhash:       0           maxlen:        0
 add:           0           remove:        0
 f_add:         0           f_remove:      0
```

Delete all flow director rules on a port:

```
testpmd> flush_flow_director 0
```

Floating VEB

The Intel® Ethernet 700 Series support a feature called "Floating VEB".

A Virtual Ethernet Bridge (VEB) is an IEEE Edge Virtual Bridging (EVB) term for functionality that allows local switching between virtual endpoints within a physical endpoint and also with an external bridge/network.

A "Floating" VEB doesn't have an uplink connection to the outside world so all switching is done internally and remains within the host. As such, this feature provides security benefits.

In addition, a Floating VEB overcomes a limitation of normal VEBs where they cannot forward packets when the physical link is down. Floating VEBs don't need to connect to the NIC port so they can still forward traffic from VF to VF even when the physical link is down.

Therefore, with this feature enabled VFs can be limited to communicating with each other but not an outside network, and they can do so even when there is no physical uplink on the associated NIC port.

To enable this feature, the user should pass a `devargs` parameter to the EAL, for example:

```
-w 84:00.0,enable_floating_veb=1
```

In this configuration the PMD will use the floating VEB feature for all the VFs created by this PF device.

Alternatively, the user can specify which VFs need to connect to this floating VEB using the `floating_veb_list` argument:

```
-w 84:00.0,enable_floating_veb=1,floating_veb_list=1;3-4
```

In this example `VF1`, `VF3` and `VF4` connect to the floating VEB, while other VFs connect to the normal VEB.

The current implementation only supports one floating VEB and one regular VEB. VFs can connect to a floating VEB or a regular VEB according to the configuration passed on the EAL command line.

The floating VEB functionality requires a NIC firmware version of 5.0 or greater.

Dynamic Device Personalization (DDP)

The Intel® Ethernet 700 Series except for the Intel Ethernet Connection X722 support a feature called "Dynamic Device Personalization (DDP)", which is used to configure hardware by downloading a profile to support protocols/filters which are not supported by default. The DDP functionality requires a NIC firmware version of 6.0 or greater.

Current implementation supports GTP-C/GTP-U/PPPoE/PPPoL2TP/ESP, steering can be used with rte_flow API.

GTPv1 package is released, and it can be downloaded from https://downloadcenter.intel.com/download/27587.

PPPoE package is released, and it can be downloaded from https://downloadcenter.intel.com/download/28040.

ESP-AH package is released, and it can be downloaded from https://downloadcenter.intel.com/download/29446.

Load a profile which supports GTP and store backup profile:

```
testpmd> ddp add 0 ./gtp.pkgo,./backup.pkgo
```

Delete a GTP profile and restore backup profile:

```
testpmd> ddp del 0 ./backup.pkgo
```

Get loaded DDP package info list:

```
testpmd> ddp get list 0
```

Display information about a GTP profile:

```
testpmd> ddp get info ./gtp.pkgo
```

Input set configuration

Input set for any PCTYPE can be configured with user defined configuration, For example, to use only 48bit prefix for IPv6 src address for IPv6 TCP RSS:

```
testpmd> port config 0 pctype 43 hash_inset clear all
testpmd> port config 0 pctype 43 hash_inset set field 13
testpmd> port config 0 pctype 43 hash_inset set field 14
testpmd> port config 0 pctype 43 hash_inset set field 15
```

Queue region configuration

The Intel® Ethernet 700 Series supports a feature of queue regions configuration for RSS in the PF, so that different traffic classes or different packet classification types can be separated to different queues in different queue regions. There is an API for configuration of queue regions in RSS with a command line. It can parse the parameters of the region index, queue number, queue start index, user priority, traffic classes and so on. Depending on commands from the command line, it will call i40e private APIs and start the process of setting or flushing the queue region configuration. As this feature is specific for i40e only private APIs are used. These new `test_pmd` commands are as shown below. For details please refer to *Testpmd Application User Guide*.

```
testpmd> set port (port_id) queue-region region_id (value) \
             queue_start_index (value) queue_num (value)
testpmd> set port (port_id) queue-region region_id (value) flowtype (value)
testpmd> set port (port_id) queue-region UP (value) region_id (value)
testpmd> set port (port_id) queue-region flush (on|off)
testpmd> show port (port_id) queue-region
```

Generic flow API

- `RSS Flow`

  RSS Flow supports to set hash input set, hash function, enable hash and configure queue region. For example: Configure queue region as queue 0, 1, 2, 3.

  ```
  testpmd> flow create 0 ingress pattern end actions rss types end \
    queues 0 1 2 3 end / end
  ```

  Enable hash and set input set for ipv4-tcp.

  ```
  testpmd> flow create 0 ingress pattern eth / ipv4 / tcp / end \
    actions rss types ipv4-tcp l3-src-only end queues end / end
  ```

  Set symmetric hash enable for flow type ipv4-tcp.

  ```
  testpmd> flow create 0 ingress pattern eth / ipv4 / tcp / end \
    actions rss types ipv4-tcp end queues end func symmetric_toeplitz / end
  ```

  Set hash function as simple xor.

  ```
  testpmd> flow create 0 ingress pattern end actions rss types end \
    queues end func simple_xor / end
  ```

9.22.8 Limitations or Known issues

MPLS packet classification

For firmware versions prior to 5.0, MPLS packets are not recognized by the NIC. The L2 Payload flow type in flow director can be used to classify MPLS packet by using a command in testpmd like:

testpmd> flow_director_filter 0 mode IP add flow l2_payload ether
0x8847 flexbytes () fwd pf queue <N> fd_id <M>

With the NIC firmware version 5.0 or greater, some limited MPLS support is added: Native MPLS (MPLS in Ethernet) skip is implemented, while no new packet type, no classification or offload are possible. With this change, L2 Payload flow type in flow director cannot be used to classify MPLS packet as with previous firmware versions. Meanwhile, the Ethertype filter can be used to classify MPLS packet by using a command in testpmd like:

testpmd> ethertype_filter 0 add mac_ignr 00:00:00:00:00:00 ethertype
0x8847 fwd queue <M>

16 Byte RX Descriptor setting on DPDK VF

Currently the VF's RX descriptor mode is decided by PF. There's no PF-VF interface for VF to request the RX descriptor mode, also no interface to notify VF its own RX descriptor mode. For all available versions of the i40e driver, these drivers don't support 16 byte RX descriptor. If the Linux i40e kernel driver is used as host driver, while DPDK i40e PMD is used as the VF driver, DPDK cannot choose 16 byte receive descriptor. The reason is that the RX descriptor is already set to 32 byte by the i40e kernel driver. That is to say, user should keep `CONFIG_RTE_LIBRTE_I40E_16BYTE_RX_DESC=n` in config file. In the future, if the Linux i40e driver supports 16 byte RX descriptor, user should make sure the DPDK VF uses the same RX descriptor mode, 16 byte or 32 byte, as the PF driver.

The same rule for DPDK PF + DPDK VF. The PF and VF should use the same RX descriptor mode. Or the VF RX will not work.

Receive packets with Ethertype 0x88A8

Due to the FW limitation, PF can receive packets with Ethertype 0x88A8 only when floating VEB is disabled.

Incorrect Rx statistics when packet is oversize

When a packet is over maximum frame size, the packet is dropped. However, the Rx statistics, when calling *rte_eth_stats_get* incorrectly shows it as received.

VF & TC max bandwidth setting

The per VF max bandwidth and per TC max bandwidth cannot be enabled in parallel. The behavior is different when handling per VF and per TC max bandwidth setting. When enabling per VF max bandwidth, SW will check if per TC max bandwidth is enabled. If so, return failure. When enabling per TC max bandwidth, SW will check if per VF max bandwidth is enabled. If so, disable per VF max bandwidth and continue with per TC max bandwidth setting.

TC TX scheduling mode setting

There are 2 TX scheduling modes for TCs, round robin and strict priority mode. If a TC is set to strict priority mode, it can consume unlimited bandwidth. It means if APP has set the max bandwidth for that TC, it comes to no effect. It's suggested to set the strict priority mode for a TC that is latency sensitive but no consuming much bandwidth.

VF performance is impacted by PCI extended tag setting

To reach maximum NIC performance in the VF the PCI extended tag must be enabled. The DPDK i40e PF driver will set this feature during initialization, but the kernel PF driver does not. So when running traffic on a VF which is managed by the kernel PF driver, a significant NIC performance downgrade has been observed (for 64 byte packets, there is about 25% line-rate downgrade for a 25GbE device and about 35% for a 40GbE device).

For kernel version >= 4.11, the kernel's PCI driver will enable the extended tag if it detects that the device supports it. So by default, this is not an issue. For kernels <= 4.11 or when the PCI extended tag is disabled it can be enabled using the steps below.

1. Get the current value of the PCI configure register:

```
setpci -s <XX:XX.X> a8.w
```

2. Set bit 8:

```
value = value | 0x100
```

3. Set the PCI configure register with new value:

```
setpci -s <XX:XX.X> a8.w=<value>
```

Vlan strip of VF

The VF vlan strip function is only supported in the i40e kernel driver >= 2.1.26.

DCB function

DCB works only when RSS is enabled.

Global configuration warning

I40E PMD will set some global registers to enable some function or set some configure. Then when using different ports of the same NIC with Linux kernel and DPDK, the port with Linux kernel will be impacted by the port with DPDK. For example, register I40E_GL_SWT_L2TAGCTRL is used to control L2 tag, i40e PMD uses I40E_GL_SWT_L2TAGCTRL to set vlan TPID. If setting TPID in port A with DPDK, then the configuration will also impact port B in the NIC with kernel driver, which don't want to use the TPID. So PMD reports warning to clarify what is changed by writing global register.

9.22.9 High Performance of Small Packets on 40GbE NIC

As there might be firmware fixes for performance enhancement in latest version of firmware image, the firmware update might be needed for getting high performance. Check the Intel support website for the latest firmware updates. Users should consult the release notes specific to a DPDK release to identify the validated firmware version for a NIC using the i40e driver.

Use 16 Bytes RX Descriptor Size

As i40e PMD supports both 16 and 32 bytes RX descriptor sizes, and 16 bytes size can provide helps to high performance of small packets. Configuration of `CONFIG_RTE_LIBRTE_I40E_16BYTE_RX_DESC` in config files can be changed to use 16 bytes size RX descriptors.

Input set requirement of each pctype for FDIR

Each PCTYPE can only have one specific FDIR input set at one time. For example, if creating 2 rte_flow rules with different input set for one PCTYPE, it will fail and return the info "Conflict with the first rule's input set", which means the current rule's input set conflicts with the first rule's. Remove the first rule if want to change the input set of the PCTYPE.

9.22.10 Example of getting best performance with l3fwd example

The following is an example of running the DPDK `l3fwd` sample application to get high performance with a server with Intel Xeon processors and Intel Ethernet CNA XL710.

The example scenario is to get best performance with two Intel Ethernet CNA XL710 40GbE ports. See Fig. 9.1 for the performance test setup.

Fig. 9.1: Performance Test Setup

1. Add two Intel Ethernet CNA XL710 to the platform, and use one port per card to get best performance. The reason for using two NICs is to overcome a PCIe v3.0 limitation since it cannot provide 80GbE bandwidth for two 40GbE ports, but two different PCIe v3.0 x8 slot can. Refer to the sample NICs output above, then we can select `82:00.0` and `85:00.0` as test ports:

```
82:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
85:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
```

2. Connect the ports to the traffic generator. For high speed testing, it's best to use a hardware traffic generator.
3. Check the PCI devices numa node (socket id) and get the cores number on the exact socket id. In this case, `82:00.0` and `85:00.0` are both in socket 1, and the cores on socket 1 in the referenced platform are 18-35 and 54-71. Note: Don't use 2 logical cores on the same core (e.g core18 has 2 logical cores, core18 and core54), instead, use 2 logical cores from different cores (e.g core18 and core19).
4. Bind these two ports to igb_uio.
5. As to Intel Ethernet CNA XL710 40GbE port, we need at least two queue pairs to achieve best performance, then two queues per port will be required, and each queue pair will need a dedicated CPU core for receiving/transmitting packets.
6. The DPDK sample application `l3fwd` will be used for performance testing, with using two ports for bi-directional forwarding. Compile the `l3fwd sample` with the default lpm mode.
7. The command line of running l3fwd would be something like the following:

   ```
   ./l3fwd -l 18-21 -n 4 -w 82:00.0 -w 85:00.0 \
           -- -p 0x3 --config '(0,0,18),(0,1,19),(1,0,20),(1,1,21)'
   ```

   This means that the application uses core 18 for port 0, queue pair 0 forwarding, core 19 for port 0, queue pair 1 forwarding, core 20 for port 1, queue pair 0 forwarding, and core 21 for port 1, queue pair 1 forwarding.
8. Configure the traffic at a traffic generator.
   - Start creating a stream on packet generator.
   - Set the Ethernet II type to 0x0800.

Tx bytes affected by the link status change

For firmware versions prior to 6.01 for X710 series and 3.33 for X722 series, the tx_bytes statistics data is affected by the link down event. Each time the link status changes to down, the tx_bytes decreases 110 bytes.

9.23 ICE Poll Mode Driver

The ice PMD (librte_pmd_ice) provides poll mode driver support for 10/25/50/100 Gbps Intel® Ethernet 810 Series Network Adapters based on the Intel Ethernet Controller E810.

9.23.1 Prerequisites

- The E810 is currently in sampling state only. To obtain early samples and/or get further information about kernel drivers, firmware and DDP support, please speak to your Intel representative.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.
- To get better performance on Intel platforms, please follow the "How to get best performance with NICs on Intel platforms" section of the *Getting Started Guide for Linux*.

9.23.2 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_ICE_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_ice` driver.

- `CONFIG_RTE_LIBRTE_ICE_DEBUG_*` (default `n`)

  Toggle display of generic debugging messages.

- `CONFIG_RTE_LIBRTE_ICE_16BYTE_RX_DESC` (default `n`)

  Toggle to use a 16-byte RX descriptor, by default the RX descriptor is 32 byte.

Runtime Config Options

- `Safe Mode Support` (default `0`)

  If driver failed to load OS package, by default driver's initialization failed. But if user intend to use the device without OS package, user can take `devargs` parameter `safe-mode-support`, for example:

  ```
  -w 80:00.0,safe-mode-support=1
  ```

  Then the driver will be initialized successfully and the device will enter Safe Mode. NOTE: In Safe mode, only very limited features are available, features like RSS, checksum, fdir, tunneling ... are all disabled.

- `Generic Flow Pipeline Mode Support` (default `0`)

  In pipeline mode, a flow can be set at one specific stage by setting parameter `priority`. Currently, we support two stages: priority = 0 or !0. Flows with priority 0 located at the first pipeline stage which typically be used as a firewall to drop the packet on a blacklist(we called it permission stage). At this stage, flow rules are created for the device's exact match engine: switch. Flows with priority !0 located at the second stage, typically packets are classified here and be steered to specific queue or queue group (we called it distribution stage), At this stage, flow rules are created for device's flow director engine. For none-pipeline mode, `priority` is ignored, a flow rule can be created as a flow director rule or a switch rule depends on its pattern/action and the resource allocation situation, all flows are virtually at the same pipeline stage. By default, generic flow API is enabled in none-pipeline mode, user can choose to use pipeline mode by setting `devargs` parameter `pipeline-mode-support`, for example:

```
-w 80:00.0,pipeline-mode-support=1
```

- `Flow Mark Support` (default `0`)

  This is a hint to the driver to select the data path that supports flow mark extraction by default. NOTE: This is an experimental devarg, it will be removed when any of below conditions is ready. 1) all data paths support flow mark (currently vPMD does not) 2) a new offload like RTE_DEV_RX_OFFLOAD_FLOW_MARK be introduced as a standard way to hint. Example:

```
-w 80:00.0,flow-mark-support=1
```

- `Protocol extraction for per queue`

  Configure the RX queues to do protocol extraction into mbuf for protocol handling acceleration, like checking the TCP SYN packets quickly.

  The argument format is:

```
-w 18:00.0,proto_xtr=<queues:protocol>[<queues:protocol>...]
-w 18:00.0,proto_xtr=<protocol>
```

  Queues are grouped by `(` and `)` within the group. The `-` character is used as a range separator and `,` is used as a single number separator. The grouping `()` can be omitted for single element group. If no queues are specified, PMD will use this protocol extraction type for all queues.

  Protocol is : `vlan, ipv4, ipv6, ipv6_flow, tcp`.

```
testpmd -w 18:00.0,proto_xtr='[(1,2-3,8-9):tcp,10-13:vlan]'
```

  This setting means queues 1, 2-3, 8-9 are TCP extraction, queues 10-13 are VLAN extraction, other queues run with no protocol extraction.

```
testpmd -w 18:00.0,proto_xtr=vlan,proto_xtr='[(1,2-3,8-9):tcp,10-23:ipv6]'
```

  This setting means queues 1, 2-3, 8-9 are TCP extraction, queues 10-23 are IPv6 extraction, other queues use the default VLAN extraction.

  The extraction metadata is copied into the registered dynamic mbuf field, and the related dynamic mbuf flags is set.

Table 9.3: Protocol extraction : `vlan`

VLAN2			VLAN1		
PCP	D	VID	PCP	D	VID

VLAN1 - single or EVLAN (first for QinQ).

VLAN2 - C-VLAN (second for QinQ).

Table 9.4: Protocol extraction : `ipv4`

IPHDR2			IPHDR1	
Ver	Hdr Len	ToS	TTL	Protocol

IPHDR1 - IPv4 header word 4, “TTL” and “Protocol” fields.

IPHDR2 - IPv4 header word 0, “Ver”, “Hdr Len” and “Type of Service” fields.

Table 9.5: Protocol extraction : `ipv6`

IPHDR2			IPHDR1	
Ver	Traffic class	Flow	Next Header	Hop Limit

IPHDR1 - IPv6 header word 3, "Next Header" and "Hop Limit" fields.

IPHDR2 - IPv6 header word 0, "Ver", "Traffic class" and high 4 bits of "Flow Label" fields.

Table 9.6: Protocol extraction : `ipv6_flow`

IPHDR2		IPHDR1
Ver	Traffic class	Flow Label

IPHDR1 - IPv6 header word 1, 16 low bits of the "Flow Label" field.

IPHDR2 - IPv6 header word 0, "Ver", "Traffic class" and high 4 bits of "Flow Label" fields.

Table 9.7: Protocol extraction : `tcp`

TCPHDR2	TCPHDR1		
Reserved	Offset	RSV	Flags

TCPHDR1 - TCP header word 6, "Data Offset" and "Flags" fields.

TCPHDR2 - Reserved

Use `rte_net_ice_dynf_proto_xtr_metadata_get` to access the protocol extraction metadata, and use `RTE_PKT_RX_DYNF_PROTO_XTR_*` to get the metadata type of `struct rte_mbuf::ol_flags`.

The `rte_net_ice_dump_proto_xtr_metadata` routine shows how to access the protocol extraction result in `struct rte_mbuf`.

9.23.3 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

9.23.4 Features

Vector PMD

Vector PMD for RX and TX path are selected automatically. The paths are chosen based on 2 conditions.

- `CPU` On the X86 platform, the driver checks if the CPU supports AVX2. If it's supported, AVX2 paths will be chosen. If not, SSE is chosen.
- `Offload features` The supported HW offload features are described in the document ice_vec.ini. If any not supported features are used, ICE vector PMD is disabled and the normal paths are chosen.

Malicious driver detection (MDD)

It's not appropriate to send a packet, if this packet's destination MAC address is just this port's MAC address. If SW tries to send such packets, HW will report a MDD event and drop the packets.

The APPs based on DPDK should avoid providing such packets.

Device Config Function (DCF)

This section demonstrates ICE DCF PMD, which shares the core module with ICE PMD and iAVF PMD.

A DCF (Device Config Function) PMD bounds to the device's trusted VF with ID 0, it can act as a sole controlling entity to exercise advance functionality (such as switch, ACL) for the rest VFs.

The DCF PMD needs to advertise and acquire DCF capability which allows DCF to send AdminQ commands that it would like to execute over to the PF and receive responses for the same from PF.

Fig. 9.2: DCF Communication flow.

1. Create the VFs:

```
echo 4 > /sys/bus/pci/devices/0000\:18\:00.0/sriov_numvfs
```

2. Enable the VF0 trust on:

```
ip link set dev enp24s0f0 vf 0 trust on
```

3. Bind the VF0, and run testpmd with 'cap=dcf' devarg:

```
testpmd -l 22-25 -n 4 -w 18:01.0,cap=dcf -- -i
```

4. Monitor the VF2 interface network traffic:

```
tcpdump -e -nn -i enp24s1f2
```

5. Create one flow to redirect the traffic to VF2 by DCF:

```
flow create 0 priority 0 ingress pattern eth / ipv4 src is 192.168.0.2 \
dst is 192.168.0.3 / end actions vf id 2 / end
```

6. Send the packet, and it should be displayed on tcpdump:

```
sendp(Ether(src='3c:fd:fe:aa:bb:78', dst='00:00:00:01:02:03')/IP(src=' \
192.168.0.2', dst="192.168.0.3")/TCP(flags='S')/Raw(load='XXXXXXXXXX'), \
iface="enp24s0f0", count=10)
```

9.23.5 Sample Application Notes

Vlan filter

Vlan filter only works when Promiscuous mode is off.

To start `testpmd`, and add vlan 10 to port 0:

```
./app/testpmd -l 0-15 -n 4 -- -i
...

testpmd> rx_vlan add 10 0
```

9.23.6 Limitations or Known issues

The Intel E810 requires a programmable pipeline package be downloaded by the driver to support normal operations. The E810 has a limited functionality built in to allow PXE boot and other use cases, but the driver must download a package file during the driver initialization stage.

The default DDP package file name is ice.pkg. For a specific NIC, the DDP package supposed to be loaded can have a filename: ice-xxxxxx.pkg, where 'xxxxxx' is the 64-bit PCIe Device Serial Number of the NIC. For example, if the NIC's device serial number is 00-CC-BB-FF-FF-AA-05-68, the device-specific DDP package filename is ice-00ccbbffffaa0568.pkg (in hex and all low case). During initialization, the driver searches in the following paths in order: /lib/firmware/updates/intel/ice/ddp and /lib/firmware/intel/ice/ddp. The corresponding device-specific DDP package will be downloaded first if the file exists. If not, then the driver tries to load the default package. The type of loaded package is stored in `ice_adapter->active_pkg_type`.

A symbolic link to the DDP package file is also ok. The same package file is used by both the kernel driver and the DPDK PMD.

limitation

Ice code released is for evaluation only currently.

9.24 IGB Poll Mode Driver

The IGB PMD (`librte_pmd_e1000`) provides poll mode driver support for Intel 1GbE nics.

9.24.1 Features

Features of the IGB PMD are:

- Multiple queues for TX and RX
- Receiver Side Scaling (RSS)
- MAC/VLAN filtering
- Packet type information
- Double VLAN

- IEEE 1588
- TSO offload
- Checksum offload
- TCP segmentation offload
- Jumbo frames supported

9.24.2 Limitations or Known issues

9.24.3 Supported Chipsets and NICs

- Intel 82576EB 10 Gigabit Ethernet Controller
- Intel 82580EB 10 Gigabit Ethernet Controller
- Intel 82580DB 10 Gigabit Ethernet Controller
- Intel Ethernet Controller I210
- Intel Ethernet Controller I350

9.25 IGC Poll Mode Driver

The IGC PMD (librte_pmd_igc) provides poll mode driver support for Foxville I225 Series Network Adapters.

- For information about I225, please refer to: *https://ark.intel.com/content/www/us/en/ark/products/series/184686/intel-ethernet-controller-i225-series.html*

9.25.1 Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_IGC_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_igc` driver.

- `CONFIG_RTE_LIBRTE_IGC_DEBUG_*` (default `n`)

  Toggle display of generic debugging messages.

Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

Supported Chipsets and NICs

Foxville LM (I225 LM): Client 2.5G LAN vPro Corporate Foxville V (I225 V): Client 2.5G LAN Consumer Foxville I (I225 I): Client 2.5G Industrial Temp Foxville V (I225 K): Client 2.5G LAN Consumer

Sample Application Notes

9.25.2 Vlan filter

VLAN stripping off only works with inner vlan. Only the outer VLAN TPID can be set to a vlan other than 0x8100.

If extend VLAN is enabled:

- The VLAN header in a packet that carries a single VLAN header is treated as the external VLAN.
- Foxville expects that any transmitted packet to have at least the external VLAN added by the software. For those packets where an external VLAN is not present, any offload that relates to inner fields to the EtherType might not be provided.
- If VLAN TX-OFFLOAD is enabled and the packet does not contain an external VLAN, the packet is dropped, and if configured, the queue from which the packet was sent is disabled.

To start `testpmd`, add vlan 10 to port, set vlan stripping off on, set extend on, set TPID of outer VLAN to 0x9100:

```
./app/testpmd -l 4-8 -- -i
...

testpmd> vlan set filter on 0
testpmd> rx_vlan add 10 0
testpmd> vlan set strip off 0
testpmd> vlan set extend on 0
testpmd> vlan set outer tpid 0x9100 0
```

9.25.3 Flow Director

The Flow Director works in receive mode to identify specific flows or sets of flows and route them to specific queues.

The Flow Director filters includes the following types:

- ether-type filter
- 2-tuple filter(destination L4 protocol and destination L4 port)
- TCP SYN filter
- RSS filter

Start `testpmd`:

```
./testpmd -l 4-8 -- i --rxq=4 --txq=4 --pkt-filter-mode=perfect --disable-rss
```

Add a rule to direct packet whose `ether-type=0x801` to queue 1:

```
testpmd> flow create 0 ingress pattern eth type is 0x801 / end actions queue index 1 / end
```

Add a rule to direct packet whose `ip-protocol=0x6(TCP), tcp_port=0x80` to queue 1:

```
testpmd> flow create 0 ingress pattern eth / ipv4 proto is 6 / tcp dst is 0x80 / end actions␣
↪queue index 1 / end
```

Add a rule to direct packet whose `ip-protocol=0x6(TCP), SYN flag is set` to queue 1:

```
testpmd> flow validate 0 ingress pattern tcp flags spec 0x02 flags mask 0x02 / end actions␣
↪queue index 1 / end
```

Add a rule to enable ipv4-udp RSS:

```
testpmd> flow create 0 ingress pattern end actions rss types ipv4-udp end / end
```

9.26 IONIC Driver

The ionic driver provides support for Pensando server adapters. It currently supports the below models:

- Naples DSC-25
- Naples DSC-100

Please visit https://pensando.io for more information.

9.26.1 Identifying the Adapter

To find if one or more Pensando PCI Ethernet devices are installed on the host, check for the PCI devices:

```
lspci -d 1dd8:
b5:00.0 Ethernet controller: Device 1dd8:1002
b6:00.0 Ethernet controller: Device 1dd8:1002
```

9.26.2 Pre-Installation Configuration

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_IONIC_PMD` (default `y`)

  Toggle compilation of ionic PMD.

9.26.3 Building DPDK

The ionic PMD driver supports UIO and VFIO, please refer to the *DPDK documentation that comes with the DPDK suite* for instructions on how to build DPDK.

9.27 IPN3KE Poll Mode Driver

The ipn3ke PMD (librte_pmd_ipn3ke) provides poll mode driver support for Intel® FPGA PAC(Programmable Acceleration Card) N3000 based on the Intel Ethernet Controller X710/XXV710 and Intel Arria 10 FPGA.

In this card, FPGA is an acceleration bridge between network interface and the Intel Ethernet Controller. Although both FPGA and Ethernet Controllers are connected to CPU with PCIe Gen3x16 Switch, all the packet RX/TX is handled by Intel Ethernet Controller. So from application point of view the data path is still the legacy Intel Ethernet Controller X710/XXV710 PMD. Besides this, users can enable more acceleration features by FPGA IP.

9.27.1 Prerequisites

- Identifying your adapter using Intel Support and get the latest NVM/FW images.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.
- To get better performance on Intel platforms, please follow the "How to get best performance with NICs on Intel platforms" section of the *Getting Started Guide for Linux*.

9.27.2 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_IPN3KE_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_ipn3ke` driver.

Runtime Config Options

- `AFU name`

  AFU name identifies which AFU is used by IPN3KE. The AFU name format is "Port|BDF", Each FPGA can be divided into four blocks at most. "Port" identifies which FPGA block the AFU bitstream belongs to, but currently only 0 IPN3KE support. "BDF" means FPGA PCIe BDF. For example:

```
--vdev 'ipn3ke_cfg0,afu=0|b3:00.0'
```

- `FPGA Acceleration list`

  For IPN3KE FPGA can provide different bitstream, different bitstream includes different Acceleration, so users need to identify which Acceleration is used. Current IPN3KE can support TM and Flow Acceleration, for example:

```
--vdev 'ipn3ke_cfg0,afu=0|b3:00.0,fpga_acc={tm|flow}'
```

- `I40e PF name list`

  Users need to bind FPGA LineSidePort to FVL PF. So I40e PF name list should be involved in startup command. For example:

```
--vdev 'ipn3ke_cfg0,afu=0|b3:00.0,fpga_acc={tm|flow},i40e_pf={0000:b1:00.0|0000:b1:00.
↪1|0000:b1:00.2|0000:b1:00.3|0000:b5:00.0|0000:b5:00.1|0000:b5:00.2|0000:b5:00.3}'
```

9.27.3 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

9.27.4 Sample Application Notes

Packet TX/RX with FPGA Pass-through image

FPGA Pass-through bitstream is original FPGA Image.

To start `testpmd`, and add I40e PF to FPGA network port:

```
./app/testpmd -l 0-15 -n 4 --vdev 'ifpga_rawdev_cfg0,ifpga=b3:00.0,port=0' --vdev 'ipn3ke_cfg0,
↪afu=0|b3:00.0,i40e_pf={0000:b1:00.0|0000:b1:00.1|0000:b1:00.2|0000:b1:00.3|0000:b5:00.
↪0|0000:b5:00.1|0000:b5:00.2|0000:b5:00.3}' -- -i --no-numa --port-topology=loop
```

HQoS and flow acceleration

HQoS and flow acceleration bitstream is used to offloading HQoS and flow classifier.

To start `testpmd`, and add I40e PF to FPGA network port, enable FPGA HQoS and Flow Acceleration:

```
./app/testpmd -l 0-15 -n 4 --vdev 'ifpga_rawdev_cfg0,ifpga=b3:00.0,port=0' --vdev 'ipn3ke_cfg0,
↪afu=0|b3:00.0,fpga_acc={tm|flow},i40e_pf={0000:b1:00.0|0000:b1:00.1|0000:b1:00.2|0000:b1:00.
↪3|0000:b5:00.0|0000:b5:00.1|0000:b5:00.2|0000:b5:00.3}' -- -i --no-numa --forward-
↪mode=macswap
```

9.27.5 Limitations or Known issues

19.05 limitation

Ipn3ke code released in 19.05 is for evaluation only.

9.28 IXGBE Driver

9.28.1 Vector PMD for IXGBE

Vector PMD uses Intel® SIMD instructions to optimize packet I/O. It improves load/store bandwidth efficiency of L1 data cache by using a wider SSE/AVX register 1 (1). The wider register gives space to hold multiple packet buffers so as to save instruction number when processing bulk of packets.

There is no change to PMD API. The RX/TX handler are the only two entries for vPMD packet I/O. They are transparently registered at runtime RX/TX execution if all condition checks pass.

1. To date, only an SSE version of IX GBE vPMD is available.

Some constraints apply as pre-conditions for specific optimizations on bulk packet transfers. The following sections explain RX and TX constraints in the vPMD.

RX Constraints

Prerequisites and Pre-conditions

The following prerequisites apply:

- To enable vPMD to work for RX, bulk allocation for Rx must be allowed.

Ensure that the following pre-conditions are satisfied:

- rxq->rx_free_thresh >= RTE_PMD_IXGBE_RX_MAX_BURST
- rxq->rx_free_thresh < rxq->nb_rx_desc
- (rxq->nb_rx_desc % rxq->rx_free_thresh) == 0
- rxq->nb_rx_desc < (IXGBE_MAX_RING_DESC - RTE_PMD_IXGBE_RX_MAX_BURST)

These conditions are checked in the code.

Scattered packets are not supported in this mode. If an incoming packet is greater than the maximum acceptable length of one "mbuf" data size (by default, the size is 2 KB), vPMD for RX would be disabled.

By default, IXGBE_MAX_RING_DESC is set to 4096 and RTE_PMD_IXGBE_RX_MAX_BURST is set to 32.

Feature not Supported by RX Vector PMD

Some features are not supported when trying to increase the throughput in vPMD. They are:

- IEEE1588
- FDIR
- Header split
- RX checksum off load

Other features are supported using optional MACRO configuration. They include:

- HW VLAN strip
- HW extend dual VLAN

To guarantee the constraint, capabilities in dev_conf.rxmode.offloads will be checked:

- DEV_RX_OFFLOAD_VLAN_STRIP
- DEV_RX_OFFLOAD_VLAN_EXTEND
- DEV_RX_OFFLOAD_CHECKSUM
- DEV_RX_OFFLOAD_HEADER_SPLIT
- dev_conf

fdir_conf->mode will also be checked.

VF Runtime Options

The following `devargs` options can be enabled at runtime. They must be passed as part of EAL arguments. For example,

```
testpmd -w af:10.0,pflink_fullchk=1 -- -i
```

- `pflink_fullchk` (default **0**)

  When calling `rte_eth_link_get_nowait()` to get VF link status, this option is used to control how VF synchronizes its status with PF's. If set, VF will not only check the PF's physical link status by reading related register, but also check the mailbox status. We call this behavior as fully checking. And checking mailbox will trigger PF's mailbox interrupt generation. If unset, the application can get the VF's link status quickly by just reading the PF's link status register, this will avoid the whole system's mailbox interrupt generation.

  `rte_eth_link_get()` will still use the mailbox method regardless of the pflink_fullchk setting.

RX Burst Size

As vPMD is focused on high throughput, it assumes that the RX burst size is equal to or greater than 32 per burst. It returns zero if using nb_pkt < 32 as the expected packet number in the receive handler.

TX Constraint

Prerequisite

The only prerequisite is related to tx_rs_thresh. The tx_rs_thresh value must be greater than or equal to RTE_PMD_IXGBE_TX_MAX_BURST, but less or equal to RTE_IXGBE_TX_MAX_FREE_BUF_SZ. Consequently, by default the tx_rs_thresh value is in the range 32 to 64.

Feature not Supported by TX Vector PMD

TX vPMD only works when offloads is set to 0

This means that it does not support any TX offload.

9.28.2 Application Programming Interface

In DPDK release v16.11 an API for ixgbe specific functions has been added to the ixgbe PMD. The declarations for the API functions are in the header `rte_pmd_ixgbe.h`.

9.28.3 Sample Application Notes

l3fwd

When running l3fwd with vPMD, there is one thing to note. In the configuration, ensure that DEV_RX_OFFLOAD_CHECKSUM in port_conf.rxmode.offloads is NOT set. Otherwise, by default, RX vPMD is disabled.

load_balancer

As in the case of l3fwd, to enable vPMD, do NOT set DEV_RX_OFFLOAD_CHECKSUM in port_conf.rxmode.offloads. In addition, for improved performance, use -bsz "(32,32),(64,64),(32,32)" in load_balancer to avoid using the default burst size of 144.

9.28.4 Limitations or Known issues

Malicious Driver Detection not Supported

The Intel x550 series NICs support a feature called MDD (Malicious Driver Detection) which checks the behavior of the VF driver. If this feature is enabled, the VF must use the advanced context descriptor correctly and set the CC (Check Context) bit. DPDK PF doesn't support MDD, but kernel PF does. We may hit problem in this scenario kernel PF + DPDK VF. If user enables MDD in kernel PF, DPDK VF will not work. Because kernel PF thinks the VF is malicious. But actually it's not. The only reason is the VF doesn't act as MDD required. There's significant performance impact to support MDD. DPDK should check if the advanced context descriptor should be set and set it. And DPDK has to ask the info about the header length from the upper layer, because parsing the packet itself is not acceptable. So, it's too expensive to support MDD. When using kernel PF + DPDK VF on x550, please make sure to use a kernel PF driver that disables MDD or can disable MDD.

Some kernel drivers already disable MDD by default while some kernels can use the command `insmod ixgbe.ko MDD=0,0` to disable MDD. Each "0" in the command refers to a port. For example, if there are 6 ixgbe ports, the command should be changed to `insmod ixgbe.ko MDD=0,0,0,0,0,0`.

Statistics

The statistics of ixgbe hardware must be polled regularly in order for it to remain consistent. Running a DPDK application without polling the statistics will cause registers on hardware to count to the maximum value, and "stick" at that value.

In order to avoid statistic registers every reaching the maximum value, read the statistics from the hardware using `rte_eth_stats_get()` or `rte_eth_xstats_get()`.

The maximum time between statistics polls that ensures consistent results can be calculated as follows:

```
max_read_interval = UINT_MAX / max_packets_per_second
max_read_interval = 4294967295 / 14880952
max_read_interval = 288.6218096127183 (seconds)
max_read_interval = ~4 mins 48 sec.
```

In order to ensure valid results, it is recommended to poll every 4 minutes.

MTU setting

Although the user can set the MTU separately on PF and VF ports, the ixgbe NIC only supports one global MTU per physical port. So when the user sets different MTUs on PF and VF ports in one physical port, the real MTU for all these PF and VF ports is the largest value set. This behavior is based on the kernel driver behavior.

VF MAC address setting

On ixgbe, the concept of "pool" can be used for different things depending on the mode. In VMDq mode, "pool" means a VMDq pool. In IOV mode, "pool" means a VF.

There is no RTE API to add a VF's MAC address from the PF. On ixgbe, the `rte_eth_dev_mac_addr_add()` function can be used to add a VF's MAC address, as a workaround.

X550 does not support legacy interrupt mode

Description

X550 cannot get interrupts if using `uio_pci_generic` module or using legacy interrupt mode of `igb_uio` or `vfio`. Because the errata of X550 states that the Interrupt Status bit is not implemented. The errata is the item #22 from X550 spec update

Implication

When using `uio_pci_generic` module or using legacy interrupt mode of `igb_uio` or `vfio`, the Interrupt Status bit would be checked if the interrupt is coming. Since the bit is not implemented in X550, the irq cannot be handled correctly and cannot report the event fd to DPDK apps. Then apps cannot get interrupts and `dmesg` will show messages like `irq #No.: `` ``nobody cared.`

Workaround

Do not bind the `uio_pci_generic` module in X550 NICs. Do not bind `igb_uio` with legacy mode in X550 NICs. Before binding `vfio` with legacy mode in X550 NICs, use `modprobe vfio `` ``nointxmask=1` to load `vfio` module if the intx is not shared with other devices.

9.28.5 Inline crypto processing support

Inline IPsec processing is supported for `RTE_SECURITY_ACTION_TYPE_INLINE_CRYPTO` mode for ESP packets only:

- ESP authentication only: AES-128-GMAC (128-bit key)
- ESP encryption and authentication: AES-128-GCM (128-bit key)

IPsec Security Gateway Sample Application supports inline IPsec processing for ixgbe PMD.

For more details see the IPsec Security Gateway Sample Application and Security library documentation.

9.28.6 Virtual Function Port Representors

The IXGBE PF PMD supports the creation of VF port representors for the control and monitoring of IXGBE virtual function devices. Each port representor corresponds to a single virtual function of that device. Using the `devargs` option `representor` the user can specify which virtual functions to create port representors for on initialization of the PF PMD by passing the VF IDs of the VFs which are required.:

```
-w DBDF,representor=[0,1,4]
```

Currently hot-plugging of representor ports is not supported so all required representors must be specified on the creation of the PF.

9.28.7 Supported Chipsets and NICs

- Intel 82599EB 10 Gigabit Ethernet Controller
- Intel 82598EB 10 Gigabit Ethernet Controller
- Intel 82599ES 10 Gigabit Ethernet Controller
- Intel 82599EN 10 Gigabit Ethernet Controller
- Intel Ethernet Controller X540-AT2
- Intel Ethernet Controller X550-BT2
- Intel Ethernet Controller X550-AT2
- Intel Ethernet Controller X550-AT
- Intel Ethernet Converged Network Adapter X520-SR1
- Intel Ethernet Converged Network Adapter X520-SR2
- Intel Ethernet Converged Network Adapter X520-LR1
- Intel Ethernet Converged Network Adapter X520-DA1
- Intel Ethernet Converged Network Adapter X520-DA2
- Intel Ethernet Converged Network Adapter X520-DA4
- Intel Ethernet Converged Network Adapter X520-QDA1
- Intel Ethernet Converged Network Adapter X520-T2
- Intel 10 Gigabit AF DA Dual Port Server Adapter
- Intel 10 Gigabit AT Server Adapter
- Intel 10 Gigabit AT2 Server Adapter
- Intel 10 Gigabit CX4 Dual Port Server Adapter
- Intel 10 Gigabit XF LR Server Adapter
- Intel 10 Gigabit XF SR Dual Port Server Adapter
- Intel 10 Gigabit XF SR Server Adapter
- Intel Ethernet Converged Network Adapter X540-T1
- Intel Ethernet Converged Network Adapter X540-T2

- Intel Ethernet Converged Network Adapter X550-T1
- Intel Ethernet Converged Network Adapter X550-T2

9.29 Intel Virtual Function Driver

Supported Intel® Ethernet Controllers (see the *DPDK Release Notes* for details) support the following modes of operation in a virtualized environment:

- **SR-IOV mode**: Involves direct assignment of part of the port resources to different guest operating systems using the PCI-SIG Single Root I/O Virtualization (SR IOV) standard, also known as "native mode" or "pass-through" mode. In this chapter, this mode is referred to as IOV mode.
- **VMDq mode**: Involves central management of the networking resources by an IO Virtual Machine (IOVM) or a Virtual Machine Monitor (VMM), also known as software switch acceleration mode. In this chapter, this mode is referred to as the Next Generation VMDq mode.

9.29.1 SR-IOV Mode Utilization in a DPDK Environment

The DPDK uses the SR-IOV feature for hardware-based I/O sharing in IOV mode. Therefore, it is possible to partition SR-IOV capability on Ethernet controller NIC resources logically and expose them to a virtual machine as a separate PCI function called a "Virtual Function". Refer to Fig. 9.3.

Therefore, a NIC is logically distributed among multiple virtual machines (as shown in Fig. 9.3), while still having global data in common to share with the Physical Function and other Virtual Functions. The DPDK fm10kvf, i40evf, igbvf or ixgbevf as a Poll Mode Driver (PMD) serves for the Intel® 82576 Gigabit Ethernet Controller, Intel® Ethernet Controller I350 family, Intel® 82599 10 Gigabit Ethernet Controller NIC, Intel® Fortville 10/40 Gigabit Ethernet Controller NIC's virtual PCI function, or PCIe host-interface of the Intel Ethernet Switch FM10000 Series. Meanwhile the DPDK Poll Mode Driver (PMD) also supports "Physical Function" of such NIC's on the host.

The DPDK PF/VF Poll Mode Driver (PMD) supports the Layer 2 switch on Intel® 82576 Gigabit Ethernet Controller, Intel® Ethernet Controller I350 family, Intel® 82599 10 Gigabit Ethernet Controller, and Intel® Fortville 10/40 Gigabit Ethernet Controller NICs so that guest can choose it for inter virtual machine traffic in SR-IOV mode.

For more detail on SR-IOV, please refer to the following documents:

- SR-IOV provides hardware based I/O sharing
- PCI-SIG-Single Root I/O Virtualization Support on IA
- Scalable I/O Virtualized Servers

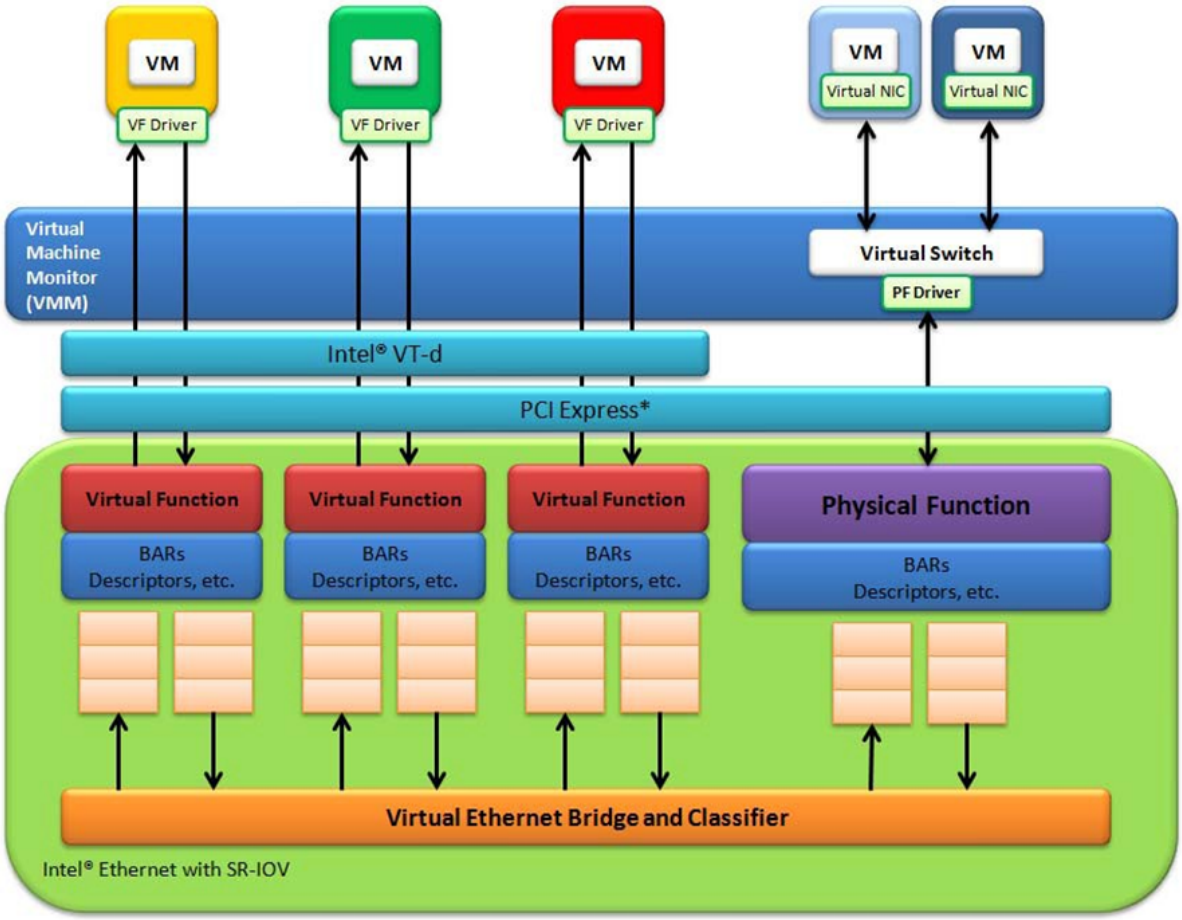


Fig. 9.3: Virtualization for a Single Port NIC in SR-IOV Mode

Physical and Virtual Function Infrastructure

The following describes the Physical Function and Virtual Functions infrastructure for the supported Ethernet Controller NICs.

Virtual Functions operate under the respective Physical Function on the same NIC Port and therefore have no access to the global NIC resources that are shared between other functions for the same NIC port.

A Virtual Function has basic access to the queue resources and control structures of the queues assigned to it. For global resource access, a Virtual Function has to send a request to the Physical Function for that port, and the Physical Function operates on the global resources on behalf of the Virtual Function. For this out-of-band communication, an SR-IOV enabled NIC provides a memory buffer for each Virtual Function, which is called a "Mailbox".

Intel® Ethernet Adaptive Virtual Function

Adaptive Virtual Function (IAVF) is a SR-IOV Virtual Function with the same device id (8086:1889) on different Intel Ethernet Controller. IAVF Driver is VF driver which supports for all future Intel devices without requiring a VM update. And since this happens to be an adaptive VF driver, every new drop of the VF driver would add more and more advanced features that can be turned on in the VM if the underlying HW device supports those advanced features based on a device agnostic way without ever compromising on the base functionality. IAVF provides generic hardware interface and interface between IAVF driver and a compliant PF driver is specified.

Intel products starting Ethernet Controller 700 Series to support Adaptive Virtual Function.

The way to generate Virtual Function is like normal, and the resource of VF assignment depends on the NIC Infrastructure.

For more detail on SR-IOV, please refer to the following documents:

- Intel® IAVF HAS

Note: To use DPDK IAVF PMD on Intel® 700 Series Ethernet Controller, the device id (0x1889) need to specified during device assignment in hypervisor. Take qemu for example, the device assignment should carry the IAVF device id (0x1889) like `-device vfio-pci,x-pci-device-id=0x1889, host=03:0a.0`.

The PCIE host-interface of Intel Ethernet Switch FM10000 Series VF infrastructure

In a virtualized environment, the programmer can enable a maximum of *64 Virtual Functions (VF)* globally per PCIE host-interface of the Intel Ethernet Switch FM10000 Series device. Each VF can have a maximum of 16 queue pairs. The Physical Function in host could be only configured by the Linux* fm10k driver (in the case of the Linux Kernel-based Virtual Machine [KVM]), DPDK PMD PF driver doesn't support it yet.

For example,

- Using Linux* fm10k driver:

```
rmmod fm10k (To remove the fm10k module)
insmod fm0k.ko max_vfs=2,2 (To enable two Virtual Functions per port)
```

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Intel® X710/XL710 Gigabit Ethernet Controller VF Infrastructure

In a virtualized environment, the programmer can enable a maximum of *128 Virtual Functions (VF)* globally per Intel® X710/XL710 Gigabit Ethernet Controller NIC device. The number of queue pairs of each VF can be configured by `CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_VF` in `config` file. The Physical Function in host could be either configured by the Linux* i40e driver (in the case of the Linux Kernel-based Virtual Machine [KVM]) or by DPDK PMD PF driver. When using both DPDK PMD PF/VF drivers, the whole NIC will be taken over by DPDK based application.

For example,

- Using Linux* i40e driver:

```
rmmod i40e (To remove the i40e module)
insmod i40e.ko max_vfs=2,2 (To enable two Virtual Functions per port)
```

- Using the DPDK PMD PF i40e driver:

  Kernel Params: iommu=pt, intel_iommu=on

```
modprobe uio
insmod igb_uio
./dpdk-devbind.py -b igb_uio bb:ss.f
echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific␣
↪PCI device)
```

  Launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

For Intel® X710/XL710 Gigabit Ethernet Controller, queues are in pairs. One queue pair means one receive queue and one transmit queue. The default number of queue pairs per VF is 4, and can be 16 in maximum.

Intel® 82599 10 Gigabit Ethernet Controller VF Infrastructure

The programmer can enable a maximum of *63 Virtual Functions* and there must be *one Physical Function* per Intel® 82599 10 Gigabit Ethernet Controller NIC port. The reason for this is that the device allows for a maximum of 128 queues per port and a virtual/physical function has to have at least one queue pair (RX/TX). The current implementation of the DPDK ixgbevf driver supports a single queue pair (RX/TX) per Virtual Function. The Physical Function in host could be either configured by the Linux* ixgbe driver (in the case of the Linux Kernel-based Virtual Machine [KVM]) or by DPDK PMD PF driver. When using both DPDK PMD PF/VF drivers, the whole NIC will be taken over by DPDK based application.

For example,

- Using Linux* ixgbe driver:

```
rmmod ixgbe (To remove the ixgbe module)
insmod ixgbe max_vfs=2,2 (To enable two Virtual Functions per port)
```

- Using the DPDK PMD PF ixgbe driver:

  Kernel Params: iommu=pt, intel_iommu=on

```
modprobe uio
insmod igb_uio
./dpdk-devbind.py -b igb_uio bb:ss.f
echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific␣
↪PCI device)
```

  Launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

- Using the DPDK PMD PF ixgbe driver to enable VF RSS:

  Same steps as above to install the modules of uio, igb_uio, specify max_vfs for PCI device, and launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

  The available queue number (at most 4) per VF depends on the total number of pool, which is determined by the max number of VF at PF initialization stage and the number of queue specified in config:

  – If the max number of VFs (max_vfs) is set in the range of 1 to 32:

    If the number of Rx queues is specified as 4 (`--rxq=4` in testpmd), then there are totally 32 pools (ETH_32_POOLS), and each VF could have 4 Rx queues;

    If the number of Rx queues is specified as 2 (`--rxq=2` in testpmd), then there are totally 32 pools (ETH_32_POOLS), and each VF could have 2 Rx queues;

- If the max number of VFs (max_vfs) is in the range of 33 to 64:

  If the number of Rx queues in specified as 4 (`--rxq=4` in testpmd), then error message is expected as `rxq` is not correct at this case;

  If the number of rxq is 2 (`--rxq=2` in testpmd), then there is totally 64 pools (ETH_64_POOLS), and each VF have 2 Rx queues;

On host, to enable VF RSS functionality, rx mq mode should be set as ETH_MQ_RX_VMDQ_RSS or ETH_MQ_RX_RSS mode, and SRIOV mode should be activated (max_vfs >= 1). It also needs config VF RSS information like hash function, RSS key, RSS key length.

Note: The limitation for VF RSS on Intel® 82599 10 Gigabit Ethernet Controller is: The hash and key are shared among PF and all VF, the RETA table with 128 entries is also shared among PF and all VF; So it could not to provide a method to query the hash and reta content per VF on guest, while, if possible, please query them on host for the shared RETA information.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Intel® 82576 Gigabit Ethernet Controller and Intel® Ethernet Controller I350 Family VF Infrastructure

In a virtualized environment, an Intel® 82576 Gigabit Ethernet Controller serves up to eight virtual machines (VMs). The controller has 16 TX and 16 RX queues. They are generally referred to (or thought of) as queue pairs (one TX and one RX queue). This gives the controller 16 queue pairs.

A pool is a group of queue pairs for assignment to the same VF, used for transmit and receive operations. The controller has eight pools, with each pool containing two queue pairs, that is, two TX and two RX queues assigned to each VF.

In a virtualized environment, an Intel® Ethernet Controller I350 family device serves up to eight virtual machines (VMs) per port. The eight queues can be accessed by eight different VMs if configured correctly (the i350 has 4x1GbE ports each with 8T X and 8 RX queues), that means, one Transmit and one Receive queue assigned to each VF.

For example,

- Using Linux* igb driver:

```
rmmod igb (To remove the igb module)
insmod igb max_vfs=2,2 (To enable two Virtual Functions per port)
```

- Using DPDK PMD PF igb driver:

  Kernel Params: iommu=pt, intel_iommu=on modprobe uio

```
insmod igb_uio
./dpdk-devbind.py -b igb_uio bb:ss.f
echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific␣
↪pci device)
```

  Launch DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a four-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence, starting from 0 to 7. However:

- Virtual Functions 0 and 4 belong to Physical Function 0
- Virtual Functions 1 and 5 belong to Physical Function 1
- Virtual Functions 2 and 6 belong to Physical Function 2
- Virtual Functions 3 and 7 belong to Physical Function 3

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Validated Hypervisors

The validated hypervisor is:

- KVM (Kernel Virtual Machine) with Qemu, version 0.14.0

However, the hypervisor is bypassed to configure the Virtual Function devices using the Mailbox interface, the solution is hypervisor-agnostic. Xen* and VMware* (when SR- IOV is supported) will also be able to support the DPDK with Virtual Function driver support.

Expected Guest Operating System in Virtual Machine

The expected guest operating systems in a virtualized environment are:

- Fedora* 14 (64-bit)
- Ubuntu* 10.04 (64-bit)

For supported kernel versions, refer to the *DPDK Release Notes*.

9.29.2 Setting Up a KVM Virtual Machine Monitor

The following describes a target environment:

- Host Operating System: Fedora 14
- Hypervisor: KVM (Kernel Virtual Machine) with Qemu version 0.14.0
- Guest Operating System: Fedora 14
- Linux Kernel Version: Refer to the *DPDK Getting Started Guide*
- Target Applications: l2fwd, l3fwd-vf

The setup procedure is as follows:

1. Before booting the Host OS, open **BIOS setup** and enable **Intel® VT features**.
2. While booting the Host OS kernel, pass the intel_iommu=on kernel command line argument using GRUB. When using DPDK PF driver on host, pass the iommu=pt kernel command line argument in GRUB.
3. Download qemu-kvm-0.14.0 from http://sourceforge.net/projects/kvm/files/qemu-kvm/ and install it in the Host OS using the following steps:

   When using a recent kernel (2.6.25+) with kvm modules included:

   ```
   tar xzf qemu-kvm-release.tar.gz
   cd qemu-kvm-release
   ./configure --prefix=/usr/local/kvm
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   When using an older kernel, or a kernel from a distribution without the kvm modules, you must download (from the same link), compile and install the modules yourself:

   ```
   tar xjf kvm-kmod-release.tar.bz2
   cd kvm-kmod-release
   ./configure
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   qemu-kvm installs in the /usr/local/bin directory.

   For more details about KVM configuration and usage, please refer to:

   http://www.linux-kvm.org/page/HOWTO1.

4. Create a Virtual Machine and install Fedora 14 on the Virtual Machine. This is referred to as the Guest Operating System (Guest OS).
5. Download and install the latest ixgbe driver from:

   http://downloadcenter.intel.com/Detail_Desc.aspx?agr=Y&DwnldID=14687

6. In the Host OS

   When using Linux kernel ixgbe driver, unload the Linux ixgbe driver and reload it with the max_vfs=2,2 argument:

```
rmmod ixgbe
modprobe ixgbe max_vfs=2,2
```

When using DPDK PMD PF driver, insert DPDK kernel module igb_uio and set the number of VF by sysfs max_vfs:

```
modprobe uio
insmod igb_uio
./dpdk-devbind.py -b igb_uio 02:00.0 02:00.1 0e:00.0 0e:00.1
echo 2 > /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
echo 2 > /sys/bus/pci/devices/0000\:02\:00.1/max_vfs
echo 2 > /sys/bus/pci/devices/0000\:0e\:00.0/max_vfs
echo 2 > /sys/bus/pci/devices/0000\:0e\:00.1/max_vfs
```

Note: You need to explicitly specify number of vfs for each port, for example, in the command above, it creates two vfs for the first two ixgbe ports.

Let say we have a machine with four physical ixgbe ports:

0000:02:00.0

0000:02:00.1

0000:0e:00.0

0000:0e:00.1

The command above creates two vfs for device 0000:02:00.0:

```
ls -alrt /sys/bus/pci/devices/0000\:02\:00.0/virt*
lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/virtfn1 -> ../
↪0000:02:10.2
lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/virtfn0 -> ../
↪0000:02:10.0
```

It also creates two vfs for device 0000:02:00.1:

```
ls -alrt /sys/bus/pci/devices/0000\:02\:00.1/virt*
lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/virtfn1 -> ../
↪0000:02:10.3
lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/virtfn0 -> ../
↪0000:02:10.1
```

7. List the PCI devices connected and notice that the Host OS shows two Physical Functions (traditional ports) and four Virtual Functions (two for each port). This is the result of the previous step.

8. Insert the pci_stub module to hold the PCI devices that are freed from the default driver using the following command (see http://www.linux-kvm.org/page/How_to_assign_devices_with_VT-d_in_KVM Section 4 for more information):

```
sudo /sbin/modprobe pci-stub
```

Unbind the default driver from the PCI devices representing the Virtual Functions. A script to perform this action is as follows:

```
echo "8086 10ed" > /sys/bus/pci/drivers/pci-stub/new_id
echo 0000:08:10.0 > /sys/bus/pci/devices/0000:08:10.0/driver/unbind
echo 0000:08:10.0 > /sys/bus/pci/drivers/pci-stub/bind
```

where, 0000:08:10.0 belongs to the Virtual Function visible in the Host OS.

9. Now, start the Virtual Machine by running the following command:

```
/usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -smp 4 -boot c -hda lucid.qcow2 -device pci-
↪assign,host=08:10.0
```

where:

— -m = memory to assign

—-smp = number of smp cores

— -boot = boot option

—-hda = virtual disk image

— -device = device to attach

Note: — The pci-assign,host=08:10.0 value indicates that you want to attach a PCI device to a Virtual Machine and the respective (Bus:Device.Function) numbers should be passed for the Virtual Function to be attached.

— qemu-kvm-0.14.0 allows a maximum of four PCI devices assigned to a VM, but this is qemu-kvm version dependent since qemu-kvm-0.14.1 allows a maximum of five PCI devices.

— qemu-system-x86_64 also has a -cpu command line option that is used to select the cpu_model to emulate in a Virtual Machine. Therefore, it can be used as:

```
/usr/local/kvm/bin/qemu-system-x86_64 -cpu ?

(to list all available cpu_models)

/usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -cpu host -smp 4 -boot c -hda lucid.qcow2 -
↪device pci-assign,host=08:10.0

(to use the same cpu_model equivalent to the host cpu)
```

For more information, please refer to: http://wiki.qemu.org/Features/CPUModels.

---

10. If use vfio-pci to pass through device instead of pci-assign, steps 8 and 9 need to be updated to bind device to vfio-pci and replace pci-assign with vfio-pci when start virtual machine.

```
sudo /sbin/modprobe vfio-pci

echo "8086 10ed" > /sys/bus/pci/drivers/vfio-pci/new_id
echo 0000:08:10.0 > /sys/bus/pci/devices/0000:08:10.0/driver/unbind
echo 0000:08:10.0 > /sys/bus/pci/drivers/vfio-pci/bind

/usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -smp 4 -boot c -hda lucid.qcow2 -device␣
↪vfio-pci,host=08:10.0
```

11. Install and run DPDK host app to take over the Physical Function. Eg.

```
make install T=x86_64-native-linux-gcc
./x86_64-native-linux-gcc/app/testpmd -l 0-3 -n 4 -- -i
```

12. Finally, access the Guest OS using vncviewer with the localhost:5900 port and check the lspci command output in the Guest OS. The virtual functions will be listed as available for use.

13. Configure and install the DPDK with an x86_64-native-linux-gcc configuration on the Guest OS as normal, that is, there is no change to the normal installation procedure.

```
make config T=x86_64-native-linux-gcc O=x86_64-native-linux-gcc
cd x86_64-native-linux-gcc
make
```

Note: If you are unable to compile the DPDK and you are getting "error: CPU you selected does not support x86-64 instruction set", power off the Guest OS and start the virtual machine with the correct -cpu option in the qemu- system-x86_64 command as shown in step 9. You must select the best x86_64 cpu_model to emulate or you can select host option if available.

Note: Run the DPDK l2fwd sample application in the Guest OS with Hugepages enabled. For the expected benchmark performance, you must pin the cores from the Guest OS to the Host OS (taskset can be used to do this) and you must also look at the PCI Bus layout on the board to ensure you are not running the traffic over the QPI Interface.

Note:

- The Virtual Machine Manager (the Fedora package name is virt-manager) is a utility for virtual machine management that can also be used to create, start, stop and delete virtual machines. If this option is used, step 2 and 6 in the instructions provided will be different.
- virsh, a command line utility for virtual machine management, can also be used to bind and unbind devices to a virtual machine in Ubuntu. If this option is used, step 6 in the instructions provided will be different.
- The Virtual Machine Monitor (see Fig. 9.4) is equivalent to a Host OS with KVM installed as described in the instructions.

9.29.3 DPDK SR-IOV PMD PF/VF Driver Usage Model

Fast Host-based Packet Processing

Software Defined Network (SDN) trends are demanding fast host-based packet handling. In a virtualization environment, the DPDK VF PMD driver performs the same throughput result as a non-VT native environment.

With such host instance fast packet processing, lots of services such as filtering, QoS, DPI can be offloaded on the host fast path.

Fig. 9.5 shows the scenario where some VMs directly communicate externally via a VFs, while others connect to a virtual switch and share the same uplink bandwidth.

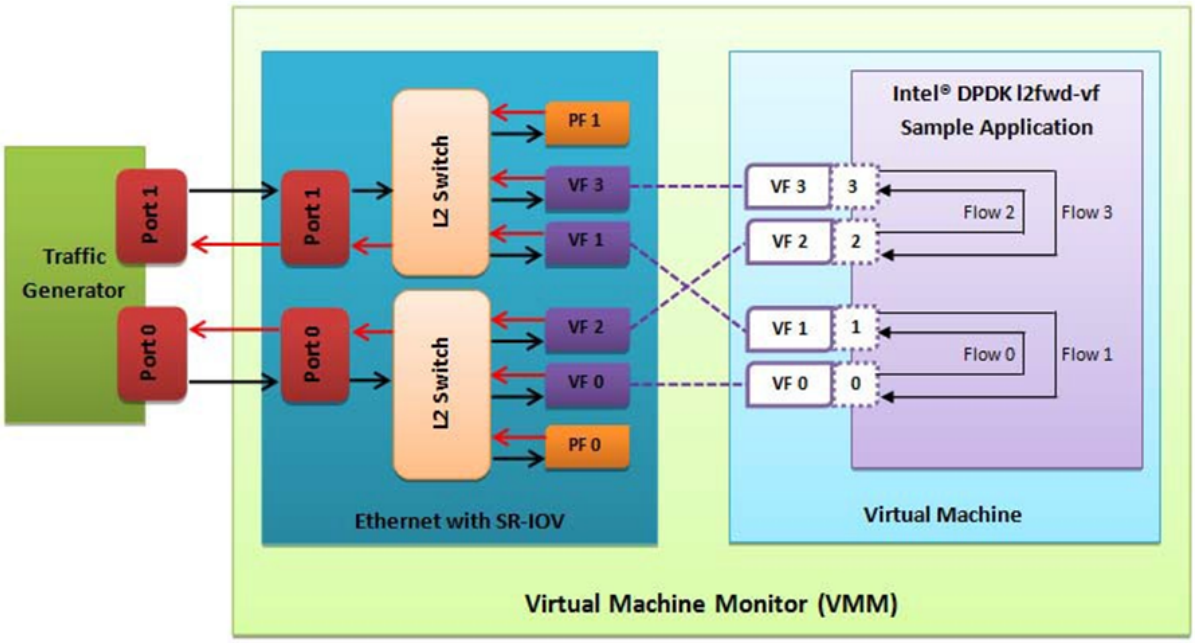


Fig. 9.4: Performance Benchmark Setup

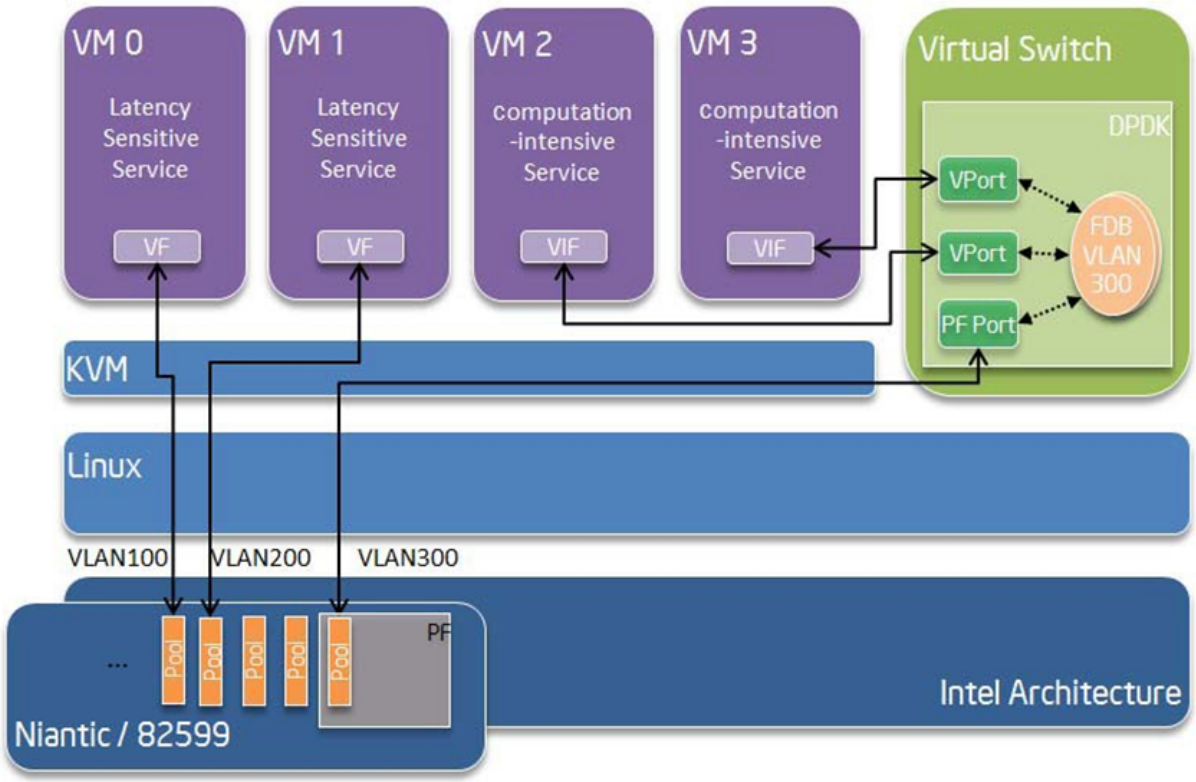


Fig. 9.5: Fast Host-based Packet Processing

9.29.4 SR-IOV (PF/VF) Approach for Inter-VM Communication

Inter-VM data communication is one of the traffic bottle necks in virtualization platforms. SR-IOV device assignment helps a VM to attach the real device, taking advantage of the bridge in the NIC. So VF-to-VF traffic within the same physical port (VM0<->VM1) have hardware acceleration. However, when VF crosses physical ports (VM0<->VM2), there is no such hardware bridge. In this case, the DPDK PMD PF driver provides host forwarding between such VMs.

Fig. 9.6 shows an example. In this case an update of the MAC address lookup tables in both the NIC and host DPDK application is required.

In the NIC, writing the destination of a MAC address belongs to another cross device VM to the PF specific pool. So when a packet comes in, its destination MAC address will match and forward to the host DPDK PMD application.

In the host DPDK application, the behavior is similar to L2 forwarding, that is, the packet is forwarded to the correct PF pool. The SR-IOV NIC switch forwards the packet to a specific VM according to the MAC destination address which belongs to the destination VF on the VM.

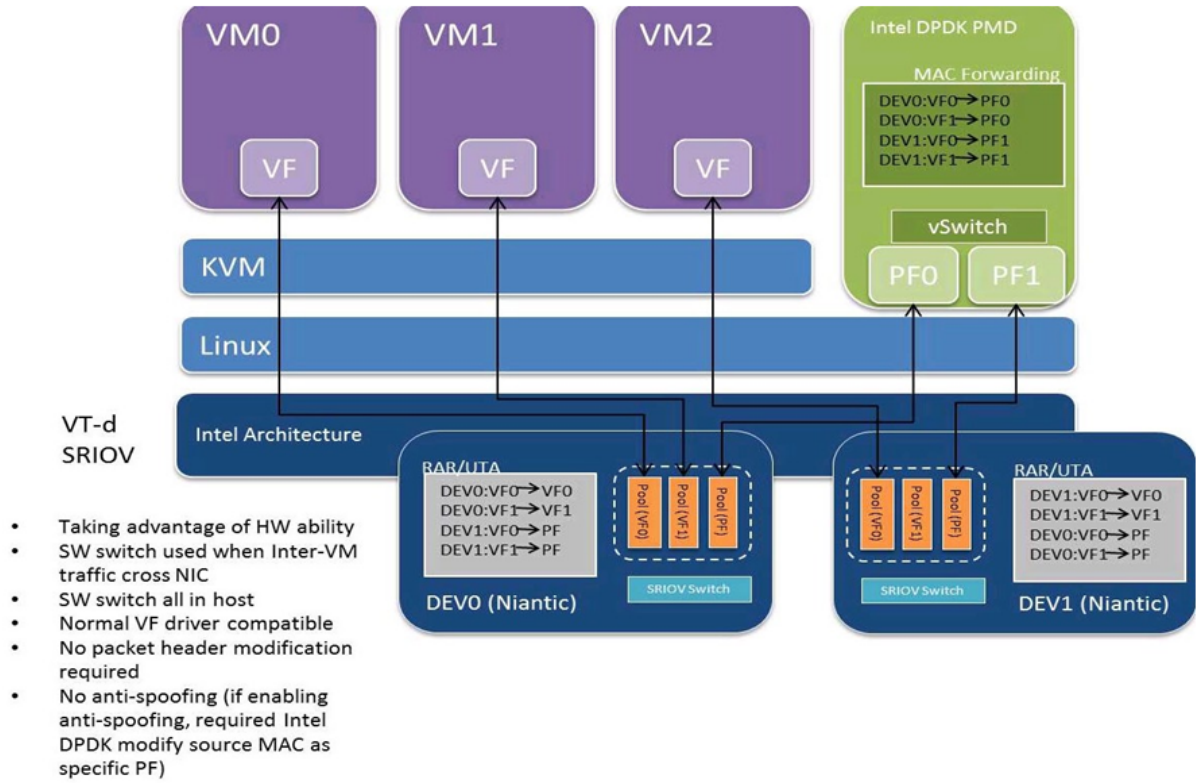


Fig. 9.6: Inter-VM Communication

9.30 KNI Poll Mode Driver

KNI PMD is wrapper to the *librte_kni* library.

This PMD enables using KNI without having a KNI specific application, any forwarding application can use PMD interface for KNI.

Sending packets to any DPDK controlled interface or sending to the Linux networking stack will be transparent to the DPDK application.

To create a KNI device `net_kni#` device name should be used, and this will create `kni#` Linux virtual network interface.

There is no physical device backend for the virtual KNI device.

Packets sent to the KNI Linux interface will be received by the DPDK application, and DPDK application may forward packets to a physical NIC or to a virtual device (like another KNI interface or PCAP interface).

To forward any traffic from physical NIC to the Linux networking stack, an application should control a physical port and create one virtual KNI port, and forward between two.

Using this PMD requires KNI kernel module be inserted.

9.30.1 Usage

EAL `--vdev` argument can be used to create KNI device instance, like:

```
testpmd --vdev=net_kni0 --vdev=net_kn1 -- -i
```

Above command will create `kni0` and `kni1` Linux network interfaces, those interfaces can be controlled by standard Linux tools.

When testpmd forwarding starts, any packets sent to `kni0` interface forwarded to the `kni1` interface and vice versa.

There is no hard limit on number of interfaces that can be created.

9.30.2 Default interface configuration

`librte_kni` can create Linux network interfaces with different features, feature set controlled by a configuration struct, and KNI PMD uses a fixed configuration:

```
Interface name: kni#
force bind kernel thread to a core : NO
mbuf size: (rte_pktmbuf_data_room_size(pktmbuf_pool) - RTE_PKTMBUF_HEADROOM)
mtu: (conf.mbuf_size - RTE_ETHER_HDR_LEN)
```

KNI control path is not supported with the PMD, since there is no physical backend device by default.

9.30.3 PMD arguments

`no_request_thread`, by default PMD creates a pthread for each KNI interface to handle Linux network interface control commands, like `ifconfig kni0 up`

With `no_request_thread` option, pthread is not created and control commands not handled by PMD.

By default request thread is enabled. And this argument should not be used most of the time, unless this PMD used with customized DPDK application to handle requests itself.

Argument usage:

```
testpmd --vdev "net_kni0,no_request_thread=1" -- -i
```

9.30.4 PMD log messages

If KNI kernel module (rte_kni.ko) not inserted, following error log printed:

```
"KNI: KNI subsystem has not been initialized. Invoke rte_kni_init() first"
```

9.30.5 PMD testing

It is possible to test PMD quickly using KNI kernel module loopback feature:

- Insert KNI kernel module with loopback support:

```
insmod build/kmod/rte_kni.ko lo_mode=lo_mode_fifo_skb
```

- Start testpmd with no physical device but two KNI virtual devices:

```
./testpmd --vdev net_kni0 --vdev net_kni1 -- -i
```

```
...
Configuring Port 0 (socket 0)
KNI: pci: 00:00:00       c580:b8
Port 0: 1A:4A:5B:7C:A2:8C
Configuring Port 1 (socket 0)
KNI: pci: 00:00:00       600:b9
Port 1: AE:95:21:07:93:DD
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

- Observe Linux interfaces

```
$ ifconfig kni0 && ifconfig kni1
kni0: flags=4098<BROADCAST,MULTICAST>  mtu 1500
        ether ae:8e:79:8e:9b:c8  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

kni1: flags=4098<BROADCAST,MULTICAST>  mtu 1500
```

(continues on next page)

(continued from previous page)

```
        ether 9e:76:43:53:3e:9b  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- Start forwarding with tx_first:

```
testpmd> start tx_first
```

- Quit and check forwarding stats:

```
testpmd> quit
Telling cores to stop...
Waiting for lcores to finish...

---------------------- Forward statistics for port 0  ----------------------
RX-packets: 35637905       RX-dropped: 0             RX-total: 35637905
TX-packets: 35637947       TX-dropped: 0             TX-total: 35637947
----------------------------------------------------------------------------

---------------------- Forward statistics for port 1  ----------------------
RX-packets: 35637915       RX-dropped: 0             RX-total: 35637915
TX-packets: 35637937       TX-dropped: 0             TX-total: 35637937
----------------------------------------------------------------------------

++++++++++++++++ Accumulated forward statistics for all ports++++++++++++++++
RX-packets: 71275820       RX-dropped: 0             RX-total: 71275820
TX-packets: 71275884       TX-dropped: 0             TX-total: 71275884
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

9.31 LiquidIO VF Poll Mode Driver

The LiquidIO VF PMD library (librte_pmd_lio) provides poll mode driver support for Cavium LiquidIO® II server adapter VFs. PF management and VF creation can be done using kernel driver.

More information can be found at Cavium Official Website.

9.31.1 Supported LiquidIO Adapters

- LiquidIO II CN2350 210SV/225SV
- LiquidIO II CN2350 210SVPT
- LiquidIO II CN2360 210SV/225SV
- LiquidIO II CN2360 210SVPT

9.31.2 Pre-Installation Configuration

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_LIO_PMD` (default `y`)

  Toggle compilation of LiquidIO PMD.

- `CONFIG_RTE_LIBRTE_LIO_DEBUG_RX` (default `n`)

  Toggle display of receive fast path run-time messages.

- `CONFIG_RTE_LIBRTE_LIO_DEBUG_TX` (default `n`)

  Toggle display of transmit fast path run-time messages.

- `CONFIG_RTE_LIBRTE_LIO_DEBUG_MBOX` (default `n`)

  Toggle display of mailbox messages.

- `CONFIG_RTE_LIBRTE_LIO_DEBUG_REGS` (default `n`)

  Toggle display of register reads and writes.

9.31.3 SR-IOV: Prerequisites and Sample Application Notes

This section provides instructions to configure SR-IOV with Linux OS.

1. Verify SR-IOV and ARI capabilities are enabled on the adapter using `lspci`:

   ```
   lspci -s <slot> -vvv
   ```

   Example output:

   ```
   [...]
   Capabilities: [148 v1] Alternative Routing-ID Interpretation (ARI)
   [...]
   Capabilities: [178 v1] Single Root I/O Virtualization (SR-IOV)
   [...]
   Kernel driver in use: LiquidIO
   ```

2. Load the kernel module:

   ```
   modprobe liquidio
   ```

3. Bring up the PF ports:

   ```
   ifconfig p4p1 up
   ifconfig p4p2 up
   ```

4. Change PF MTU if required:

   ```
   ifconfig p4p1 mtu 9000
   ifconfig p4p2 mtu 9000
   ```

5. Create VF device(s):

   Echo number of VFs to be created into `"sriov_numvfs"` sysfs entry of the parent PF.

```
echo 1 > /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000:03:00.1/sriov_numvfs
```

6. Assign VF MAC address:

   Assign MAC address to the VF using iproute2 utility. The syntax is:

```
ip link set <PF iface> vf <VF id> mac <macaddr>
```

   Example output:

```
ip link set p4p1 vf 0 mac F2:A8:1B:5E:B4:66
```

7. Assign VF(s) to VM.

   The VF devices may be passed through to the guest VM using qemu or virt-manager or virsh etc.

   Example qemu guest launch command:

```
./qemu-system-x86_64 -name lio-vm -machine accel=kvm \
-cpu host -m 4096 -smp 4 \
-drive file=<disk_file>,if=none,id=disk1,format=<type> \
-device virtio-blk-pci,scsi=off,drive=disk1,id=virtio-disk1,bootindex=1 \
-device vfio-pci,host=03:00.3 -device vfio-pci,host=03:08.3
```

8. Running testpmd

   Refer to the document *compiling and testing a PMD for a NIC* to run `testpmd` application.

   Note: Use `igb_uio` instead of `vfio-pci` in VM.

   Example output:

```
[...]
EAL: PCI device 0000:03:00.3 on NUMA socket 0
EAL:   probe driver: 177d:9712 net_liovf
EAL:   using IOMMU type 1 (Type 1)
PMD: net_liovf[03:00.3]INFO: DEVICE : CN23XX VF
EAL: PCI device 0000:03:08.3 on NUMA socket 0
EAL:   probe driver: 177d:9712 net_liovf
PMD: net_liovf[03:08.3]INFO: DEVICE : CN23XX VF
Interactive-mode selected
USER1: create a new mbuf pool <mbuf_pool_socket_0>: n=171456, size=2176, socket=0
Configuring Port 0 (socket 0)
PMD: net_liovf[03:00.3]INFO: Starting port 0
Port 0: F2:A8:1B:5E:B4:66
Configuring Port 1 (socket 0)
PMD: net_liovf[03:08.3]INFO: Starting port 1
Port 1: 32:76:CC:EE:56:D7
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

9. Enabling VF promiscuous mode

   One VF per PF can be marked as trusted for promiscuous mode.

```
ip link set dev <PF iface> vf <VF id> trust on
```

9.31.4 Limitations

VF MTU

VF MTU is limited by PF MTU. Raise PF value before configuring VF for larger packet size.

VLAN offload

Tx VLAN insertion is not supported and consequently VLAN offload feature is marked partial.

Ring size

Number of descriptors for Rx/Tx ring should be in the range 128 to 512.

CRC stripping

LiquidIO adapters strip ethernet FCS of every packet coming to the host interface.

9.32 Memif Poll Mode Driver

Shared memory packet interface (memif) PMD allows for DPDK and any other client using memif (DPDK, VPP, libmemif) to communicate using shared memory. Memif is Linux only.

The created device transmits packets in a raw format. It can be used with Ethernet mode, IP mode, or Punt/Inject. At this moment, only Ethernet mode is supported in DPDK memif implementation.

Memif works in two roles: master and slave. Slave connects to master over an existing socket. It is also a producer of shared memory file and initializes the shared memory. Each interface can be connected to one peer interface at same time. The peer interface is identified by id parameter. Master creates the socket and listens for any slave connection requests. The socket may already exist on the system. Be sure to remove any such sockets, if you are creating a master interface, or you will see an "Address already in use" error. Function `rte_pmd_memif_remove()`, which removes memif interface, will also remove a listener socket, if it is not being used by any other interface.

The method to enable one or more interfaces is to use the `--vdev=net_memif0` option on the DPDK application command line. Each `--vdev=net_memif1` option given will create an interface named net_memif0, net_memif1, and so on. Memif uses unix domain socket to transmit control messages. Each memif has a unique id per socket. This id is used to identify peer interface. If you are connecting multiple interfaces using same socket, be sure to specify unique ids `id=0`, `id=1`, etc. Note that if you assign a socket to a master interface it becomes a listener socket. Listener socket can not be used by a slave interface on same client.

Table 9.8: **Memif configuration options**

Option	Description	Default	Valid value
id=0	Used to identify peer interface	0	uint32_t
role=master	Set memif role	slave	master|slave
bsize=1024	Size of single packet buffer	2048	uint16_t
rsize=11	Log2 of ring size. If rsize is 10, actual ring size is 1024	10	1-14
socket=/tmp/memif.sock	Socket filename	/tmp/memif.sock	string len 108
mac=01:23:45:ab:cd:ef	Mac address	01:ab:23:cd:45:ef	
secret=abc123	Secret is an optional security option, which if specified, must be matched by peer		string len 24
zero-copy=yes	Enable/disable zero-copy slave mode. Only relevant to slave, requires '–single-file-segments' eal argument	no	yes|no

Connection establishment

In order to create memif connection, two memif interfaces, each in separate process, are needed. One interface in `master` role and other in `slave` role. It is not possible to connect two interfaces in a single process. Each interface can be connected to one interface at same time, identified by matching id parameter.

Memif driver uses unix domain socket to exchange required information between memif interfaces. Socket file path is specified at interface creation see *Memif configuration options* table above. If socket is used by `master` interface, it's marked as listener socket (in scope of current process) and listens to connection requests from other processes. One socket can be used by multiple interfaces. One process can have `slave` and `master` interfaces at the same time, provided each role is assigned unique socket.

For detailed information on memif control messages, see: net/memif/memif.h.

Slave interface attempts to make a connection on assigned socket. Process listening on this socket will extract the connection request and create a new connected socket (control channel). Then it sends the 'hello' message (`MEMIF_MSG_TYPE_HELLO`), containing configuration boundaries. Slave interface adjusts its configuration accordingly, and sends 'init' message (`MEMIF_MSG_TYPE_INIT`). This message among others contains interface id. Driver uses this id to find master interface, and assigns the control channel to this interface. If such interface is found, 'ack' message (`MEMIF_MSG_TYPE_ACK`) is sent. Slave interface sends 'add region' message (`MEMIF_MSG_TYPE_ADD_REGION`) for every region allocated. Master responds to each of these messages with 'ack' message. Same behavior applies to rings. Slave sends 'add ring' message (`MEMIF_MSG_TYPE_ADD_RING`) for every initialized ring. Master again responds to each message with 'ack' message. To finalize the connection, slave interface sends 'connect' message (`MEMIF_MSG_TYPE_CONNECT`). Upon receiving this message master maps regions to its address space, initializes rings and responds with 'connected' message (`MEMIF_MSG_TYPE_CONNECTED`). Disconnect (`MEMIF_MSG_TYPE_DISCONNECT`) can be sent by both master and slave interfaces at any time, due to driver error or if the interface is being deleted.

Files

- net/memif/memif.h - *control messages definitions*
- net/memif/memif_socket.h
- net/memif/memif_socket.c

9.32.1 Shared memory

Shared memory format

Slave is producer and master is consumer. Memory regions, are mapped shared memory files, created by memif slave and provided to master at connection establishment. Regions contain rings and buffers. Rings and buffers can also be separated into multiple regions. For no-zero-copy, rings and buffers are stored inside single memory region to reduce the number of opened files.

region n (no-zero-copy):

Rings		Buffers		
S2M rings	M2S rings	packet buffer 0	.	pb ((1 << pmd->run.log2_ring_size)*(s2m + m2s))-1

S2M OR M2S Rings:

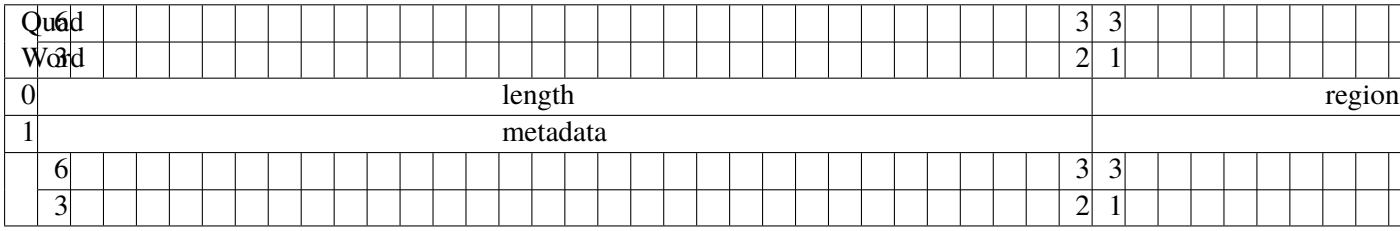
ring 0	ring 1	ring num_s2m_rings - 1

ring 0:

ring header	(1 << pmd->run.log2_ring_size) * desc

Descriptors are assigned packet buffers in order of rings creation. If we have one ring in each direction and ring size is 1024, then first 1024 buffers will belong to S2M ring and last 1024 will belong to M2S ring. In case of zero-copy, buffers are dequeued and enqueued as needed.

Descriptor format



Flags field - flags (Quad Word 0, bits 0:15)

Bits	Name	Functionality
0	MEMIF_DESC_FLAG_NEXT	Is chained buffer. When set, the packet is divided into multiple buffers. May not be contiguous.

Region index - region (Quad Word 0, 16:31)

Index of memory region, the buffer is located in.

Data length - length (Quad Word 0, 32:63)

Length of transmitted/received data.

Data Offset - offset (Quad Word 1, 0:31)

Data start offset from memory region address. *.regions[desc->region].addr + desc->offset*

Metadata - metadata (Quad Word 1, 32:63)

Buffer metadata.

Files

- net/memif/memif.h - *descriptor and ring definitions*
- net/memif/rte_eth_memif.c - *eth_memif_rx() eth_memif_tx()*

9.32.2 Zero-copy slave

Zero-copy slave can be enabled with memif configuration option 'zero-copy=yes'. This option is only relevant to slave and requires eal argument '–single-file-segments'. This limitation is in place, because it is too expensive to identify memseg for each packet buffer, resulting in worse performance than with zero-copy disabled. With single file segments we can calculate offset from the beginning of the file for each packet buffer.

Shared memory format

Region 0 is created by memif driver and contains rings. Slave interface exposes DPDK memory (memseg). Instead of using memfd_create() to create new shared file, existing memsegs are used. Master interface functions the same as with zero-copy disabled.

region 0:

Rings	
S2M rings	M2S rings

region n:

Buffers
memseg

Buffers are dequeued and enqueued as needed. Offset descriptor field is calculated at tx. Only single file segments mode (EAL option –single-file-segments) is supported, as calculating offset from multiple segments is too expensive.

Example: testpmd

In this example we run two instances of testpmd application and transmit packets over memif.

First create `master` interface:

```
#./build/app/testpmd -l 0-1 --proc-type=primary --file-prefix=pmd1 --vdev=net_memif,
↪role=master -- -i
```

Now create `slave` interface (master must be already running so the slave will connect):

```
#./build/app/testpmd -l 2-3 --proc-type=primary --file-prefix=pmd2 --vdev=net_memif -- -i
```

You can also enable `zero-copy` on `slave` interface:

```
#./build/app/testpmd -l 2-3 --proc-type=primary --file-prefix=pmd2 --vdev=net_memif,zero-
↪copy=yes --single-file-segments -- -i
```

Start forwarding packets:

```
Slave:
    testpmd> start

Master:
    testpmd> start tx_first
```

Show status:

```
testpmd> show port stats 0
```

For more details on testpmd please refer to *Testpmd Application User Guide*.

Example: testpmd and VPP

For information on how to get and run VPP please see https://wiki.fd.io/view/VPP.

Start VPP in interactive mode (should be by default). Create memif master interface in VPP:

```
vpp# create interface memif id 0 master no-zero-copy
vpp# set interface state memif0/0 up
vpp# set interface ip address memif0/0 192.168.1.1/24
```

To see socket filename use show memif command:

```
vpp# show memif
sockets
 id  listener    filename
  0   yes (1)     /run/vpp/memif.sock
...
```

Now create memif interface by running testpmd with these command line options:

```
#./testpmd --vdev=net_memif,socket=/run/vpp/memif.sock -- -i
```

Testpmd should now create memif slave interface and try to connect to master. In testpmd set forward option to icmpecho and start forwarding:

```
testpmd> set fwd icmpecho
testpmd> start
```

Send ping from VPP:

```
vpp# ping 192.168.1.2
64 bytes from 192.168.1.2: icmp_seq=2 ttl=254 time=36.2918 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=254 time=23.3927 ms
64 bytes from 192.168.1.2: icmp_seq=4 ttl=254 time=24.2975 ms
64 bytes from 192.168.1.2: icmp_seq=5 ttl=254 time=17.7049 ms
```

Example: testpmd memif loopback

In this example we will create 2 memif ports connected into loopback. The situation is analogous to cross connecting 2 ports of the NIC by cable.

To set the loopback, just use the same socket and id with different roles:

```
#./testpmd --vdev=net_memif0,role=master,id=0 --vdev=net_memif1,role=slave,id=0 -- -i
```

Then start the communication:

```
testpmd> start tx_first
```

Finally we can check port stats to see the traffic:

```
testpmd> show port stats all
testpmd> show port stats all
```

9.33 MLX4 poll mode driver library

The MLX4 poll mode driver library (**librte_pmd_mlx4**) implements support for **Mellanox ConnectX-3** and **Mellanox ConnectX-3 Pro** 10/40 Gbps adapters as well as their virtual functions (VF) in SR-IOV context.

Information and documentation about this family of adapters can be found on the Mellanox website. Help is also provided by the Mellanox community.

There is also a section dedicated to this poll mode driver.

Note: Due to external dependencies, this driver is disabled by default. It must be enabled manually by setting `CONFIG_RTE_LIBRTE_MLX4_PMD=y` and recompiling DPDK.

9.33.1 Implementation details

Most Mellanox ConnectX-3 devices provide two ports but expose a single PCI bus address, thus unlike most drivers, librte_pmd_mlx4 registers itself as a PCI driver that allocates one Ethernet device per detected port.

For this reason, one cannot white/blacklist a single port without also white/blacklisting the others on the same device.

Besides its dependency on libibverbs (that implies libmlx4 and associated kernel support), librte_pmd_mlx4 relies heavily on system calls for control operations such as querying/updating the MTU and flow control parameters.

For security reasons and robustness, this driver only deals with virtual memory addresses. The way resources allocations are handled by the kernel combined with hardware specifications that allow it to handle virtual memory addresses directly ensure that DPDK applications cannot access random physical memory (or memory that does not belong to the current process).

This capability allows the PMD to coexist with kernel network interfaces which remain functional, although they stop receiving unicast packets as long as they share the same MAC address.

The *Flow isolated mode* is supported.

Compiling librte_pmd_mlx4 causes DPDK to be linked against libibverbs.

9.33.2 Configuration

Compilation options

These options can be modified in the `.config` file.

- `CONFIG_RTE_LIBRTE_MLX4_PMD` (default **n**)

  Toggle compilation of librte_pmd_mlx4 itself.

- `CONFIG_RTE_IBVERBS_LINK_DLOPEN` (default **n**)

  Build PMD with additional code to make it loadable without hard dependencies on **libibverbs** nor **libmlx4**, which may not be installed on the target system.

  In this mode, their presence is still required for it to run properly, however their absence won't prevent a DPDK application from starting (with `CONFIG_RTE_BUILD_SHARED_LIB` disabled) and they won't show up as missing with `ldd(1)`.

  It works by moving these dependencies to a purpose-built rdma-core "glue" plug-in which must either be installed in a directory whose name is based on `CONFIG_RTE_EAL_PMD_PATH` suffixed with `-glue` if set, or in a standard location for the dynamic linker (e.g. `/lib`) if left to the default empty string (`""`).

  This option has no performance impact.

- `CONFIG_RTE_IBVERBS_LINK_STATIC` (default **n**)

  Embed static flavor of the dependencies **libibverbs** and **libmlx4** in the PMD shared library or the executable static binary.

- `CONFIG_RTE_LIBRTE_MLX4_DEBUG` (default **n**)

  Toggle debugging code and stricter compilation flags. Enabling this option adds additional run-time checks and debugging messages at the cost of lower performance.

This option is available in meson:

- `ibverbs_link` can be `static`, `shared`, or `dlopen`.

Environment variables

- `MLX4_GLUE_PATH`

  A list of directories in which to search for the rdma-core "glue" plug-in, separated by colons or semi-colons.

  Only matters when compiled with `CONFIG_RTE_IBVERBS_LINK_DLOPEN` enabled and most useful when `CONFIG_RTE_EAL_PMD_PATH` is also set, since `LD_LIBRARY_PATH` has no effect in this case.

Run-time configuration

- librte_pmd_mlx4 brings kernel network interfaces up during initialization because it is affected by their state. Forcing them down prevents packets reception.
- **ethtool** operations on related kernel interfaces also affect the PMD.
- `port` parameter [int]

  This parameter provides a physical port to probe and can be specified multiple times for additional ports. All ports are probed by default if left unspecified.

- `mr_ext_memseg_en` parameter [int]

  A nonzero value enables extending memseg when registering DMA memory. If enabled, the number of entries in MR (Memory Region) lookup table on datapath is minimized and it benefits performance. On the other hand, it worsens memory utilization because registered memory is pinned by kernel driver. Even if a page in the extended chunk is freed, that doesn't become reusable until the entire memory is freed.

  Enabled by default.

Kernel module parameters

The **mlx4_core** kernel module has several parameters that affect the behavior and/or the performance of librte_pmd_mlx4. Some of them are described below.

- **num_vfs** (integer or triplet, optionally prefixed by device address strings)

  Create the given number of VFs on the specified devices.

- **log_num_mgm_entry_size** (integer)

  Device-managed flow steering (DMFS) is required by DPDK applications. It is enabled by using a negative value, the last four bits of which have a special meaning.

  - **-1**: force device-managed flow steering (DMFS).
  - **-7**: configure optimized steering mode to improve performance with the following limitation: VLAN filtering is not supported with this mode. This is the recommended mode in case VLAN filter is not needed.

9.33.3 Limitations

- For secondary process:
  - Forked secondary process not supported.
  - External memory unregistered in EAL memseg list cannot be used for DMA unless such memory has been registered by `mlx4_mr_update_ext_mp()` in primary process and remapped to the same virtual address in secondary process. If the external memory is registered by primary process but has different virtual address in secondary process, unexpected error may happen.
- CRC stripping is supported by default and always reported as "true". The ability to enable/disable CRC stripping requires OFED version 4.3-1.5.0.0 and above or rdma-core version v18 and above.
- TSO (Transmit Segmentation Offload) is supported in OFED version 4.4 and above.

9.33.4 Prerequisites

This driver relies on external libraries and kernel drivers for resources allocations and initialization. The following dependencies are not part of DPDK and must be installed separately:

- **libibverbs** (provided by rdma-core package)

  User space verbs framework used by librte_pmd_mlx4. This library provides a generic interface between the kernel and low-level user space drivers such as libmlx4.

  It allows slow and privileged operations (context initialization, hardware resources allocations) to be managed by the kernel and fast operations to never leave user space.

- **libmlx4** (provided by rdma-core package)

  Low-level user space driver library for Mellanox ConnectX-3 devices, it is automatically loaded by libibverbs.

  This library basically implements send/receive calls to the hardware queues.

- **Kernel modules**

  They provide the kernel-side verbs API and low level device drivers that manage actual hardware initialization and resources sharing with user space processes.

  Unlike most other PMDs, these modules must remain loaded and bound to their devices:

  - mlx4_core: hardware driver managing Mellanox ConnectX-3 devices.
  - mlx4_en: Ethernet device driver that provides kernel network interfaces.
  - mlx4_ib: InifiniBand device driver.
  - ib_uverbs: user space driver for verbs (entry point for libibverbs).

- **Firmware update**

  Mellanox OFED releases include firmware updates for ConnectX-3 adapters.

  Because each release provides new features, these updates must be applied to match the kernel modules and libraries they come with.

Note: Both libraries are BSD and GPL licensed. Linux kernel modules are GPL licensed.

Depending on system constraints and user preferences either RDMA core library with a recent enough Linux kernel release (recommended) or Mellanox OFED, which provides compatibility with older releases.

Current RDMA core package and Linux kernel (recommended)

- Minimal Linux kernel version: 4.14.
- Minimal RDMA core version: v15 (see RDMA core installation documentation).
- Starting with rdma-core v21, static libraries can be built:

```
cd build
CFLAGS=-fPIC cmake -DIN_PLACE=1 -DENABLE_STATIC=1 -GNinja ..
ninja
```

If rdma-core libraries are built but not installed, DPDK makefile can link them, thanks to these environment variables:

- `EXTRA_CFLAGS=-I/path/to/rdma-core/build/include`
- `EXTRA_LDFLAGS=-L/path/to/rdma-core/build/lib`
- `PKG_CONFIG_PATH=/path/to/rdma-core/build/lib/pkgconfig`

Mellanox OFED as a fallback

- Mellanox OFED version: **4.4, 4.5, 4.6**.
- firmware version: **2.42.5000** and above.

Note: Several versions of Mellanox OFED are available. Installing the version this DPDK release was developed and tested against is strongly recommended. Please check the *prerequisites*.

Installing Mellanox OFED

1. Download latest Mellanox OFED.
2. Install the required libraries and kernel modules either by installing only the required set, or by installing the entire Mellanox OFED:

   For bare metal use:

   ```
   ./mlnxofedinstall --dpdk --upstream-libs
   ```

   For SR-IOV hypervisors use:

   ```
   ./mlnxofedinstall --dpdk --upstream-libs --enable-sriov --hypervisor
   ```

   For SR-IOV virtual machine use:

   ```
   ./mlnxofedinstall --dpdk --upstream-libs --guest
   ```

3. Verify the firmware is the correct one:

   ```
   ibv_devinfo
   ```

4. Set all ports links to Ethernet, follow instructions on the screen:

   ```
   connectx_port_config
   ```

5. Continue with *section 2 of the Quick Start Guide*.

9.33.5 Quick Start Guide

1. Set all ports links to Ethernet:

```
PCI=<NIC PCI address>
echo eth > "/sys/bus/pci/devices/$PCI/mlx4_port0"
echo eth > "/sys/bus/pci/devices/$PCI/mlx4_port1"
```

Note: If using Mellanox OFED one can permanently set the port link to Ethernet using connectx_port_config tool provided by it. *Mellanox OFED as a fallback*:

2. In case of bare metal or hypervisor, configure optimized steering mode by adding the following line to `/etc/modprobe.d/mlx4_core.conf`:

```
options mlx4_core log_num_mgm_entry_size=-7
```

Note: If VLAN filtering is used, set log_num_mgm_entry_size=-1. Performance degradation can occur on this case.

3. Restart the driver:

```
/etc/init.d/openibd restart
```

or:

```
service openibd restart
```

4. Compile DPDK and you are ready to go. See instructions on *Development Kit Build System*

9.33.6 Performance tuning

1. Verify the optimized steering mode is configured:

```
cat /sys/module/mlx4_core/parameters/log_num_mgm_entry_size
```

2. Use the CPU near local NUMA node to which the PCIe adapter is connected, for better performance. For VMs, verify that the right CPU and NUMA node are pinned according to the above. Run:

```
lstopo-no-graphics
```

to identify the NUMA node to which the PCIe adapter is connected.

3. If more than one adapter is used, and root complex capabilities allow to put both adapters on the same NUMA node without PCI bandwidth degradation, it is recommended to locate both adapters on the same NUMA node. This in order to forward packets from one to the other without NUMA performance penalty.

4. Disable pause frames:

```
ethtool -A <netdev> rx off tx off
```

5. Verify IO non-posted prefetch is disabled by default. This can be checked via the BIOS configuration. Please contact you server provider for more information about the settings.

Note: On some machines, depends on the machine integrator, it is beneficial to set the PCI max read request parameter to 1K. This can be done in the following way:

To query the read request size use:

```
setpci -s <NIC PCI address> 68.w
```

If the output is different than 3XXX, set it by:

```
setpci -s <NIC PCI address> 68.w=3XXX
```

The XXX can be different on different systems. Make sure to configure according to the setpci output.

---

6. To minimize overhead of searching Memory Regions:
   - '–socket-mem' is recommended to pin memory by predictable amount.
   - Configure per-lcore cache when creating Mempools for packet buffer.
   - Refrain from dynamically allocating/freeing memory in run-time.

9.33.7 Usage example

This section demonstrates how to launch **testpmd** with Mellanox ConnectX-3 devices managed by librte_pmd_mlx4.

1. Load the kernel modules:

   ```
   modprobe -a ib_uverbs mlx4_en mlx4_core mlx4_ib
   ```

   Alternatively if MLNX_OFED is fully installed, the following script can be run:

   ```
   /etc/init.d/openibd restart
   ```

   Note: User space I/O kernel modules (uio and igb_uio) are not used and do not have to be loaded.

   ---

2. Make sure Ethernet interfaces are in working order and linked to kernel verbs. Related sysfs entries should be present:

   ```
   ls -d /sys/class/net/*/device/infiniband_verbs/uverbs* | cut -d / -f 5
   ```

   Example output:

   ```
   eth2
   eth3
   eth4
   eth5
   ```

3. Optionally, retrieve their PCI bus addresses for whitelisting:

```
{
    for intf in eth2 eth3 eth4 eth5;
    do
        (cd "/sys/class/net/${intf}/device/" && pwd -P);
    done;
} |
sed -n 's,.*/\(.*\),-w \1,p'
```

Example output:

```
-w 0000:83:00.0
-w 0000:83:00.0
-w 0000:84:00.0
-w 0000:84:00.0
```

Note: There are only two distinct PCI bus addresses because the Mellanox ConnectX-3 adapters installed on this system are dual port.

4. Request huge pages:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages/nr_hugepages
```

5. Start testpmd with basic parameters:

```
testpmd -l 8-15 -n 4 -w 0000:83:00.0 -w 0000:84:00.0 -- --rxq=2 --txq=2 -i
```

Example output:

```
[...]
EAL: PCI device 0000:83:00.0 on NUMA socket 1
EAL:   probe driver: 15b3:1007 librte_pmd_mlx4
PMD: librte_pmd_mlx4: PCI information matches, using device "mlx4_0" (VF: false)
PMD: librte_pmd_mlx4: 2 port(s) detected
PMD: librte_pmd_mlx4: port 1 MAC address is 00:02:c9:b5:b7:50
PMD: librte_pmd_mlx4: port 2 MAC address is 00:02:c9:b5:b7:51
EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL:   probe driver: 15b3:1007 librte_pmd_mlx4
PMD: librte_pmd_mlx4: PCI information matches, using device "mlx4_1" (VF: false)
PMD: librte_pmd_mlx4: 2 port(s) detected
PMD: librte_pmd_mlx4: port 1 MAC address is 00:02:c9:b5:ba:b0
PMD: librte_pmd_mlx4: port 2 MAC address is 00:02:c9:b5:ba:b1
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: librte_pmd_mlx4: 0x867d60: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867d60: RX queues number update: 0 -> 2
Port 0: 00:02:C9:B5:B7:50
Configuring Port 1 (socket 0)
PMD: librte_pmd_mlx4: 0x867da0: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867da0: RX queues number update: 0 -> 2
Port 1: 00:02:C9:B5:B7:51
Configuring Port 2 (socket 0)
PMD: librte_pmd_mlx4: 0x867de0: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867de0: RX queues number update: 0 -> 2
Port 2: 00:02:C9:B5:BA:B0
Configuring Port 3 (socket 0)
PMD: librte_pmd_mlx4: 0x867e20: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867e20: RX queues number update: 0 -> 2
Port 3: 00:02:C9:B5:BA:B1
```

(continues on next page)

(continued from previous page)

```
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 40000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex
Port 3 Link Up - speed 40000 Mbps - full-duplex
Done
testpmd>
```

9.34 MLX5 poll mode driver

The MLX5 poll mode driver library (**librte_pmd_mlx5**) provides support for **Mellanox ConnectX-4**, **Mellanox ConnectX-4 Lx** , **Mellanox ConnectX-5**, **Mellanox ConnectX-6**, **Mellanox ConnectX-6 Dx** and **Mellanox BlueField** families of 10/25/40/50/100/200 Gb/s adapters as well as their virtual functions (VF) in SR-IOV context.

Information and documentation about these adapters can be found on the Mellanox website. Help is also provided by the Mellanox community.

There is also a section dedicated to this poll mode driver.

Note: Due to external dependencies, this driver is disabled in default configuration of the "make" build. It can be enabled with `CONFIG_RTE_LIBRTE_MLX5_PMD=y` or by using "meson" build system which will detect dependencies.

9.34.1 Design

Besides its dependency on libibverbs (that implies libmlx5 and associated kernel support), librte_pmd_mlx5 relies heavily on system calls for control operations such as querying/updating the MTU and flow control parameters.

For security reasons and robustness, this driver only deals with virtual memory addresses. The way resources allocations are handled by the kernel, combined with hardware specifications that allow to handle virtual memory addresses directly, ensure that DPDK applications cannot access random physical memory (or memory that does not belong to the current process).

This capability allows the PMD to coexist with kernel network interfaces which remain functional, although they stop receiving unicast packets as long as they share the same MAC address. This means legacy linux control tools (for example: ethtool, ifconfig and more) can operate on the same network interfaces that owned by the DPDK application.

The PMD can use libibverbs and libmlx5 to access the device firmware or directly the hardware components. There are different levels of objects and bypassing abilities to get the best performances:

- Verbs is a complete high-level generic API
- Direct Verbs is a device-specific API
- DevX allows to access firmware objects
- Direct Rules manages flow steering at low-level hardware layer

Enabling librte_pmd_mlx5 causes DPDK applications to be linked against libibverbs.

9.34.2 Features

- Multi arch support: x86_64, POWER8, ARMv8, i686.
- Multiple TX and RX queues.
- Support for scattered TX and RX frames.
- IPv4, IPv6, TCPv4, TCPv6, UDPv4 and UDPv6 RSS on any number of queues.
- RSS using different combinations of fields: L3 only, L4 only or both, and source only, destination only or both.
- Several RSS hash keys, one for each flow type.
- Default RSS operation with no hash key specification.
- Configurable RETA table.
- Link flow control (pause frame).
- Support for multiple MAC addresses.
- VLAN filtering.
- RX VLAN stripping.
- TX VLAN insertion.
- RX CRC stripping configuration.
- Promiscuous mode on PF and VF.
- Multicast promiscuous mode on PF and VF.
- Hardware checksum offloads.
- Flow director (RTE_FDIR_MODE_PERFECT, RTE_FDIR_MODE_PERFECT_MAC_VLAN and RTE_ETH_FDIR_REJECT).
- Flow API, including *Flow isolated mode*.
- Multiple process.
- KVM and VMware ESX SR-IOV modes are supported.
- RSS hash result is supported.
- Hardware TSO for generic IP or UDP tunnel, including VXLAN and GRE.
- Hardware checksum Tx offload for generic IP or UDP tunnel, including VXLAN and GRE.
- RX interrupts.
- Statistics query including Basic, Extended and per queue.
- Rx HW timestamp.
- Tunnel types: VXLAN, L3 VXLAN, VXLAN-GPE, GRE, MPLSoGRE, MPLSoUDP, IP-in-IP, Geneve, GTP.
- Tunnel HW offloads: packet type, inner/outer RSS, IP and UDP checksum verification.
- NIC HW offloads: encapsulation (vxlan, gre, mplsoudp, mplsogre), NAT, routing, TTL increment/decrement, count, drop, mark. For details please see *Supported hardware offloads*.

- Flow insertion rate of more then million flows per second, when using Direct Rules.
- Support for multiple rte_flow groups.
- Per packet no-inline hint flag to disable packet data copying into Tx descriptors.
- Hardware LRO.
- Hairpin.

9.34.3 Limitations

- For secondary process:
  - Forked secondary process not supported.
  - External memory unregistered in EAL memseg list cannot be used for DMA unless such memory has been registered by `mlx5_mr_update_ext_mp()` in primary process and remapped to the same virtual address in secondary process. If the external memory is registered by primary process but has different virtual address in secondary process, unexpected error may happen.
- When using Verbs flow engine (`dv_flow_en` = 0), flow pattern without any specific VLAN will match for VLAN packets as well:

  When VLAN spec is not specified in the pattern, the matching rule will be created with VLAN as a wild card. Meaning, the flow rule:

  ```
  flow create 0 ingress pattern eth / vlan vid is 3 / ipv4 / end ...
  ```

  Will only match vlan packets with vid=3. and the flow rule:

  ```
  flow create 0 ingress pattern eth / ipv4 / end ...
  ```

  Will match any ipv4 packet (VLAN included).
- When using DV flow engine (`dv_flow_en` = 1), flow pattern without VLAN item will match untagged packets only. The flow rule:

  ```
  flow create 0 ingress pattern eth / ipv4 / end ...
  ```

  Will match untagged packets only. The flow rule:

  ```
  flow create 0 ingress pattern eth / vlan / ipv4 / end ...
  ```

  Will match tagged packets only, with any VLAN ID value. The flow rule:

  ```
  flow create 0 ingress pattern eth / vlan vid is 3 / ipv4 / end ...
  ```

  Will only match tagged packets with VLAN ID 3.
- VLAN pop offload command:
  - Flow rules having a VLAN pop offload command as one of their actions and are lacking a match on VLAN as one of their items are not supported.
  - The command is not supported on egress traffic.
- VLAN push offload is not supported on ingress traffic.

- VLAN set PCP offload is not supported on existing headers.
- A multi segment packet must have not more segments than reported by dev_infos_get() in tx_desc_lim.nb_seg_max field. This value depends on maximal supported Tx descriptor size and `txq_inline_min` settings and may be from 2 (worst case forced by maximal inline settings) to 58.
- Flows with a VXLAN Network Identifier equal (or ends to be equal) to 0 are not supported.
- L3 VXLAN and VXLAN-GPE tunnels cannot be supported together with MPLSoGRE and MPLSoUDP.
- Match on Geneve header supports the following fields only:
  - VNI
  - OAM
  - protocol type
  - options length Currently, the only supported options length value is 0.
- VF: flow rules created on VF devices can only match traffic targeted at the configured MAC addresses (see `rte_eth_dev_mac_addr_add()`).
- Match on GTP tunnel header item supports the following fields only:
  - v_pt_rsv_flags: E flag, S flag, PN flag
  - msg_type
  - teid
- No Tx metadata go to the E-Switch steering domain for the Flow group 0. The flows within group 0 and set metadata action are rejected by hardware.

Note: MAC addresses not already present in the bridge table of the associated kernel network device will be added and cleaned up by the PMD when closing the device. In case of ungraceful program termination, some entries may remain present and should be removed manually by other means.

- When Multi-Packet Rx queue is configured (`mprq_en`), a Rx packet can be externally attached to a user-provided mbuf with having EXT_ATTACHED_MBUF in ol_flags. As the mempool for the external buffer is managed by PMD, all the Rx mbufs must be freed before the device is closed. Otherwise, the mempool of the external buffers will be freed by PMD and the application which still holds the external buffers may be corrupted.
- If Multi-Packet Rx queue is configured (`mprq_en`) and Rx CQE compression is enabled (`rxq_cqe_comp_en`) at the same time, RSS hash result is not fully supported. Some Rx packets may not have PKT_RX_RSS_HASH.
- IPv6 Multicast messages are not supported on VM, while promiscuous mode and allmulticast mode are both set to off. To receive IPv6 Multicast messages on VM, explicitly set the relevant MAC address using rte_eth_dev_mac_addr_add() API.
- To support a mixed traffic pattern (some buffers from local host memory, some buffers from other devices) with high bandwidth, a mbuf flag is used.

  An application hints the PMD whether or not it should try to inline the given mbuf data buffer. PMD should do the best effort to act upon this request.

The hint flag `RTE_PMD_MLX5_FINE_GRANULARITY_INLINE` is dynamic, registered by application with rte_mbuf_dynflag_register(). This flag is purely driver-specific and declared in PMD specific header `rte_pmd_mlx5.h`, which is intended to be used by the application.

To query the supported specific flags in runtime, the function `rte_pmd_mlx5_get_dyn_flag_names` returns the array of currently (over present hardware and configuration) supported specific flags. The "not inline hint" feature operating flow is the following one:

  - application starts
  - probe the devices, ports are created
  - query the port capabilities
  - if port supporting the feature is found
  - register dynamic flag `RTE_PMD_MLX5_FINE_GRANULARITY_INLINE`
  - application starts the ports
  - on `dev_start()` PMD checks whether the feature flag is registered and enables the feature support in datapath
  - application might set the registered flag bit in `ol_flags` field of mbuf being sent and PMD will handle ones appropriately.

- The amount of descriptors in Tx queue may be limited by data inline settings. Inline data require the more descriptor building blocks and overall block amount may exceed the hardware supported limits. The application should reduce the requested Tx size or adjust data inline settings with `txq_inline_max` and `txq_inline_mpw` devargs keys.
- E-Switch decapsulation Flow:
  - can be applied to PF port only.
  - must specify VF port action (packet redirection from PF to VF).
  - optionally may specify tunnel inner source and destination MAC addresses.
- E-Switch encapsulation Flow:
  - can be applied to VF ports only.
  - must specify PF port action (packet redirection from VF to PF).
- Raw encapsulation:
  - The input buffer, used as outer header, is not validated.
- Raw decapsulation:
  - The decapsulation is always done up to the outermost tunnel detected by the HW.
  - The input buffer, providing the removal size, is not validated.
  - The buffer size must match the length of the headers to be removed.
- ICMP/ICMP6 code/type matching, IP-in-IP and MPLS flow matching are all mutually exclusive features which cannot be supported together (see *Firmware configuration*).
- LRO:
  - Requires DevX and DV flow to be enabled.

- KEEP_CRC offload cannot be supported with LRO.
- The first mbuf length, without head-room, must be big enough to include the TCP header (122B).
- Rx queue with LRO offload enabled, receiving a non-LRO packet, can forward it with size limited to max LRO size, not to max RX packet length.
- **LRO can be used with outer header of TCP packets of the standard format:**
  eth (with or without vlan) / ipv4 or ipv6 / tcp / payload

  Other TCP packets (e.g. with MPLS label) received on Rx queue with LRO enabled, will be received with bad checksum.

9.34.4 Statistics

MLX5 supports various methods to report statistics:

Port statistics can be queried using `rte_eth_stats_get()`. The received and sent statistics are through SW only and counts the number of packets received or sent successfully by the PMD. The imissed counter is the amount of packets that could not be delivered to SW because a queue was full. Packets not received due to congestion in the bus or on the NIC can be queried via the rx_discards_phy xstats counter.

Extended statistics can be queried using `rte_eth_xstats_get()`. The extended statistics expose a wider set of counters counted by the device. The extended port statistics counts the number of packets received or sent successfully by the port. As Mellanox NICs are using the *Bifurcated Linux Driver* those counters counts also packet received or sent by the Linux kernel. The counters with `_phy` suffix counts the total events on the physical port, therefore not valid for VF.

Finally per-flow statistics can by queried using `rte_flow_query` when attaching a count action for specific flow. The flow counter counts the number of packets received successfully by the port and match the specific flow.

9.34.5 Configuration

Compilation options

These options can be modified in the `.config` file.

- `CONFIG_RTE_LIBRTE_MLX5_PMD` (default **n**)

  Toggle compilation of librte_pmd_mlx5 itself.

- `CONFIG_RTE_IBVERBS_LINK_DLOPEN` (default **n**)

  Build PMD with additional code to make it loadable without hard dependencies on **libibverbs** nor **libmlx5**, which may not be installed on the target system.

  In this mode, their presence is still required for it to run properly, however their absence won't prevent a DPDK application from starting (with `CONFIG_RTE_BUILD_SHARED_LIB` disabled) and they won't show up as missing with `ldd(1)`.

  It works by moving these dependencies to a purpose-built rdma-core "glue" plug-in which must either be installed in a directory whose name is based on `CONFIG_RTE_EAL_PMD_PATH` suffixed with `-glue` if set, or in a standard location for the dynamic linker (e.g. `/lib`) if left to the default empty string ("").

This option has no performance impact.

- `CONFIG_RTE_IBVERBS_LINK_STATIC` (default **n**)

  Embed static flavor of the dependencies **libibverbs** and **libmlx5** in the PMD shared library or the executable static binary.

- `CONFIG_RTE_LIBRTE_MLX5_DEBUG` (default **n**)

  Toggle debugging code and stricter compilation flags. Enabling this option adds additional run-time checks and debugging messages at the cost of lower performance.

Note: For BlueField, target should be set to `arm64-bluefield-linux-gcc`. This will enable `CONFIG_RTE_LIBRTE_MLX5_PMD` and set `RTE_CACHE_LINE_SIZE` to 64. Default armv8a configuration of make build and meson build set it to 128 then brings performance degradation.

This option is available in meson:

- `ibverbs_link` can be `static`, `shared`, or `dlopen`.

Environment variables

- `MLX5_GLUE_PATH`

  A list of directories in which to search for the rdma-core "glue" plug-in, separated by colons or semi-colons.

  Only matters when compiled with `CONFIG_RTE_IBVERBS_LINK_DLOPEN` enabled and most useful when `CONFIG_RTE_EAL_PMD_PATH` is also set, since `LD_LIBRARY_PATH` has no effect in this case.

- `MLX5_SHUT_UP_BF`

  Configures HW Tx doorbell register as IO-mapped.

  By default, the HW Tx doorbell is configured as a write-combining register. The register would be flushed to HW usually when the write-combining buffer becomes full, but it depends on CPU design.

  Except for vectorized Tx burst routines, a write memory barrier is enforced after updating the register so that the update can be immediately visible to HW.

  When vectorized Tx burst is called, the barrier is set only if the burst size is not aligned to MLX5_VPMD_TX_MAX_BURST. However, setting this environmental variable will bring better latency even though the maximum throughput can slightly decline.

Run-time configuration

- librte_pmd_mlx5 brings kernel network interfaces up during initialization because it is affected by their state. Forcing them down prevents packets reception.
- **ethtool** operations on related kernel interfaces also affect the PMD.

Run as non-root

In order to run as a non-root user, some capabilities must be granted to the application:

```
setcap cap_sys_admin,cap_net_admin,cap_net_raw,cap_ipc_lock+ep <dpdk-app>
```

Below are the reasons of the need for each capability:

cap_sys_admin
: When using physical addresses (PA mode), with Linux >= 4.0, for access to `/proc/self/pagemap`.

cap_net_admin
: For device configuration.

cap_net_raw
: For raw ethernet queue allocation through kernel driver.

cap_ipc_lock
: For DMA memory pinning.

Driver options

- `rxq_cqe_comp_en` parameter [int]

  A nonzero value enables the compression of CQE on RX side. This feature allows to save PCI bandwidth and improve performance. Enabled by default.

  Supported on:

  - x86_64 with ConnectX-4, ConnectX-4 Lx, ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField.
  - POWER9 and ARMv8 with ConnectX-4 Lx, ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField.

- `rxq_cqe_pad_en` parameter [int]

  A nonzero value enables 128B padding of CQE on RX side. The size of CQE is aligned with the size of a cacheline of the core. If cacheline size is 128B, the CQE size is configured to be 128B even though the device writes only 64B data on the cacheline. This is to avoid unnecessary cache invalidation by device's two consecutive writes on to one cacheline. However in some architecture, it is more beneficial to update entire cacheline with padding the rest 64B rather than striding because read-modify-write could drop performance a lot. On the other hand, writing extra data will consume more PCIe bandwidth and could also drop the maximum throughput. It is recommended to empirically set this parameter. Disabled by default.

  Supported on:

  - CPU having 128B cacheline with ConnectX-5 and BlueField.

- `rxq_pkt_pad_en` parameter [int]

  A nonzero value enables padding Rx packet to the size of cacheline on PCI transaction. This feature would waste PCI bandwidth but could improve performance by avoiding partial cacheline write which may cause costly read-modify-copy in memory transaction on some architectures. Disabled by default.

Supported on:

- x86_64 with ConnectX-4, ConnectX-4 Lx, ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField.
- POWER8 and ARMv8 with ConnectX-4 Lx, ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField.

- `mprq_en` parameter [int]

  A nonzero value enables configuring Multi-Packet Rx queues. Rx queue is configured as Multi-Packet RQ if the total number of Rx queues is `rxqs_min_mprq` or more. Disabled by default.

  Multi-Packet Rx Queue (MPRQ a.k.a Striding RQ) can further save PCIe bandwidth by posting a single large buffer for multiple packets. Instead of posting a buffers per a packet, one large buffer is posted in order to receive multiple packets on the buffer. A MPRQ buffer consists of multiple fixed-size strides and each stride receives one packet. MPRQ can improve throughput for small-packet traffic.

  When MPRQ is enabled, max_rx_pkt_len can be larger than the size of user-provided mbuf even if DEV_RX_OFFLOAD_SCATTER isn't enabled. PMD will configure large stride size enough to accommodate max_rx_pkt_len as long as device allows. Note that this can waste system memory compared to enabling Rx scatter and multi-segment packet.

- `mprq_log_stride_num` parameter [int]

  Log 2 of the number of strides for Multi-Packet Rx queue. Configuring more strides can reduce PCIe traffic further. If configured value is not in the range of device capability, the default value will be set with a warning message. The default value is 4 which is 16 strides per a buffer, valid only if `mprq_en` is set.

  The size of Rx queue should be bigger than the number of strides.

- `mprq_log_stride_size` parameter [int]

  Log 2 of the size of a stride for Multi-Packet Rx queue. Configuring a smaller stride size can save some memory and reduce probability of a depletion of all available strides due to unreleased packets by an application. If configured value is not in the range of device capability, the default value will be set with a warning message. The default value is 11 which is 2048 bytes per a stride, valid only if `mprq_en` is set. With `mprq_log_stride_size` set it is possible for a pcaket to span across multiple strides. This mode allows support of jumbo frames (9K) with MPRQ. The memcopy of some packets (or part of a packet if Rx scatter is configured) may be required in case there is no space left for a head room at the end of a stride which incurs some performance penalty.

- `mprq_max_memcpy_len` parameter [int]

  The maximum length of packet to memcpy in case of Multi-Packet Rx queue. Rx packet is mem-copied to a user-provided mbuf if the size of Rx packet is less than or equal to this parameter. Otherwise, PMD will attach the Rx packet to the mbuf by external buffer attachment - `rte_pktmbuf_attach_extbuf()`. A mempool for external buffers will be allocated and managed by PMD. If Rx packet is externally attached, ol_flags field of the mbuf will have EXT_ATTACHED_MBUF and this flag must be preserved. `RTE_MBUF_HAS_EXTBUF()` checks the flag. The default value is 128, valid only if `mprq_en` is set.

- `rxqs_min_mprq` parameter [int]

  Configure Rx queues as Multi-Packet RQ if the total number of Rx queues is greater or equal to this value. The default value is 12, valid only if `mprq_en` is set.

- `txq_inline` parameter [int]

  Amount of data to be inlined during TX operations. This parameter is deprecated and converted to the new parameter `txq_inline_max` providing partial compatibility.

- `txqs_min_inline` parameter [int]

  Enable inline data send only when the number of TX queues is greater or equal to this value.

  This option should be used in combination with `txq_inline_max` and `txq_inline_mpw` below and does not affect `txq_inline_min` settings above.

  If this option is not specified the default value 16 is used for BlueField and 8 for other platforms

  The data inlining consumes the CPU cycles, so this option is intended to auto enable inline data if we have enough Tx queues, which means we have enough CPU cores and PCI bandwidth is getting more critical and CPU is not supposed to be bottleneck anymore.

  The copying data into WQE improves latency and can improve PPS performance when PCI back pressure is detected and may be useful for scenarios involving heavy traffic on many queues.

  Because additional software logic is necessary to handle this mode, this option should be used with care, as it may lower performance when back pressure is not expected.

  If inline data are enabled it may affect the maximal size of Tx queue in descriptors because the inline data increase the descriptor size and queue size limits supported by hardware may be exceeded.

- `txq_inline_min` parameter [int]

  Minimal amount of data to be inlined into WQE during Tx operations. NICs may require this minimal data amount to operate correctly. The exact value may depend on NIC operation mode, requested offloads, etc. It is strongly recommended to omit this parameter and use the default values. Anyway, applications using this parameter should take into consideration that specifying an inconsistent value may prevent the NIC from sending packets.

  If `txq_inline_min` key is present the specified value (may be aligned by the driver in order not to exceed the limits and provide better descriptor space utilization) will be used by the driver and it is guaranteed that requested amount of data bytes are inlined into the WQE beside other inline settings. This key also may update `txq_inline_max` value (default or specified explicitly in devargs) to reserve the space for inline data.

  If `txq_inline_min` key is not present, the value may be queried by the driver from the NIC via DevX if this feature is available. If there is no DevX enabled/supported the value 18 (supposing L2 header including VLAN) is set for ConnectX-4 and ConnectX-4 Lx, and 0 is set by default for ConnectX-5 and newer NICs. If packet is shorter the `txq_inline_min` value, the entire packet is inlined.

  For ConnectX-4 NIC, driver does not allow specifying value below 18 (minimal L2 header, including VLAN), error will be raised.

  For ConnectX-4 Lx NIC, it is allowed to specify values below 18, but it is not recommended and may prevent NIC from sending packets over some configurations.

  Please, note, this minimal data inlining disengages eMPW feature (Enhanced Multi-Packet Write), because last one does not support partial packet inlining. This is not very critical due to minimal data inlining is mostly required by ConnectX-4 and ConnectX-4 Lx, these NICs do not support eMPW feature.

- `txq_inline_max` parameter [int]

Specifies the maximal packet length to be completely inlined into WQE Ethernet Segment for ordinary SEND method. If packet is larger than specified value, the packet data won't be copied by the driver at all, data buffer is addressed with a pointer. If packet length is less or equal all packet data will be copied into WQE. This may improve PCI bandwidth utilization for short packets significantly but requires the extra CPU cycles.

The data inline feature is controlled by number of Tx queues, if number of Tx queues is larger than `txqs_min_inline` key parameter, the inline feature is engaged, if there are not enough Tx queues (which means not enough CPU cores and CPU resources are scarce), data inline is not performed by the driver. Assigning `txqs_min_inline` with zero always enables the data inline.

The default `txq_inline_max` value is 290. The specified value may be adjusted by the driver in order not to exceed the limit (930 bytes) and to provide better WQE space filling without gaps, the adjustment is reflected in the debug log. Also, the default value (290) may be decreased in run-time if the large transmit queue size is requested and hardware does not support enough descriptor amount, in this case warning is emitted. If `txq_inline_max` key is specified and requested inline settings can not be satisfied then error will be raised.

- `txq_inline_mpw` parameter [int]

  Specifies the maximal packet length to be completely inlined into WQE for Enhanced MPW method. If packet is large the specified value, the packet data won't be copied, and data buffer is addressed with pointer. If packet length is less or equal, all packet data will be copied into WQE. This may improve PCI bandwidth utilization for short packets significantly but requires the extra CPU cycles.

  The data inline feature is controlled by number of TX queues, if number of Tx queues is larger than `txqs_min_inline` key parameter, the inline feature is engaged, if there are not enough Tx queues (which means not enough CPU cores and CPU resources are scarce), data inline is not performed by the driver. Assigning `txqs_min_inline` with zero always enables the data inline.

  The default `txq_inline_mpw` value is 268. The specified value may be adjusted by the driver in order not to exceed the limit (930 bytes) and to provide better WQE space filling without gaps, the adjustment is reflected in the debug log. Due to multiple packets may be included to the same WQE with Enhanced Multi Packet Write Method and overall WQE size is limited it is not recommended to specify large values for the `txq_inline_mpw`. Also, the default value (268) may be decreased in run-time if the large transmit queue size is requested and hardware does not support enough descriptor amount, in this case warning is emitted. If `txq_inline_mpw` key is specified and requested inline settings can not be satisfied then error will be raised.

- `txqs_max_vec` parameter [int]

  Enable vectorized Tx only when the number of TX queues is less than or equal to this value. This parameter is deprecated and ignored, kept for compatibility issue to not prevent driver from probing.

- `txq_mpw_hdr_dseg_en` parameter [int]

  A nonzero value enables including two pointers in the first block of TX descriptor. The parameter is deprecated and ignored, kept for compatibility issue.

- `txq_max_inline_len` parameter [int]

  Maximum size of packet to be inlined. This limits the size of packet to be inlined. If the size of a packet is larger than configured value, the packet isn't inlined even though there's enough space remained in the descriptor. Instead, the packet is included with pointer. This parameter is

deprecated and converted directly to `txq_inline_mpw` providing full compatibility. Valid only if eMPW feature is engaged.

- `txq_mpw_en` parameter [int]

  A nonzero value enables Enhanced Multi-Packet Write (eMPW) for ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField. eMPW allows the TX burst function to pack up multiple packets in a single descriptor session in order to save PCI bandwidth and improve performance at the cost of a slightly higher CPU usage. When `txq_inline_mpw` is set along with `txq_mpw_en`, TX burst function copies entire packet data on to TX descriptor instead of including pointer of packet.

  The Enhanced Multi-Packet Write feature is enabled by default if NIC supports it, can be disabled by explicit specifying 0 value for `txq_mpw_en` option. Also, if minimal data inlining is requested by non-zero `txq_inline_min` option or reported by the NIC, the eMPW feature is disengaged.

- `tx_db_nc` parameter [int]

  The rdma core library can map doorbell register in two ways, depending on the environment variable "MLX5_SHUT_UP_BF":

  – As regular cached memory (usually with write combining attribute), if the variable is either missing or set to zero.

  – As non-cached memory, if the variable is present and set to not "0" value.

  The type of mapping may slightly affect the Tx performance, the optimal choice is strongly relied on the host architecture and should be deduced practically.

  If `tx_db_nc` is set to zero, the doorbell is forced to be mapped to regular memory (with write combining), the PMD will perform the extra write memory barrier after writing to doorbell, it might increase the needed CPU clocks per packet to send, but latency might be improved.

  If `tx_db_nc` is set to one, the doorbell is forced to be mapped to non cached memory, the PMD will not perform the extra write memory barrier after writing to doorbell, on some architectures it might improve the performance.

  If `tx_db_nc` is set to two, the doorbell is forced to be mapped to regular memory, the PMD will use heuristics to decide whether write memory barrier should be performed. For bursts with size multiple of recommended one (64 pkts) it is supposed the next burst is coming and no need to issue the extra memory barrier (it is supposed to be issued in the next coming burst, at least after descriptor writing). It might increase latency (on some hosts till next packets transmit) and should be used with care.

  If `tx_db_nc` is omitted or set to zero, the preset (if any) environment variable "MLX5_SHUT_UP_BF" value is used. If there is no "MLX5_SHUT_UP_BF", the default `tx_db_nc` value is zero for ARM64 hosts and one for others.

- `tx_vec_en` parameter [int]

  A nonzero value enables Tx vector on ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField NICs if the number of global Tx queues on the port is less than `txqs_max_vec`. The parameter is deprecated and ignored.

- `rx_vec_en` parameter [int]

  A nonzero value enables Rx vector if the port is not configured in multi-segment otherwise this parameter is ignored.

  Enabled by default.

- `vf_nl_en` parameter [int]

  A nonzero value enables Netlink requests from the VF to add/remove MAC addresses or/and enable/disable promiscuous/all multicast on the Netdevice. Otherwise the relevant configuration must be run with Linux iproute2 tools. This is a prerequisite to receive this kind of traffic.

  Enabled by default, valid only on VF devices ignored otherwise.

- `l3_vxlan_en` parameter [int]

  A nonzero value allows L3 VXLAN and VXLAN-GPE flow creation. To enable L3 VXLAN or VXLAN-GPE, users has to configure firmware and enable this parameter. This is a prerequisite to receive this kind of traffic.

  Disabled by default.

- `dv_xmeta_en` parameter [int]

  A nonzero value enables extensive flow metadata support if device is capable and driver supports it. This can enable extensive support of `MARK` and `META` item of `rte_flow`. The newly introduced `SET_TAG` and `SET_META` actions do not depend on `dv_xmeta_en`.

  There are some possible configurations, depending on parameter value:

  – 0, this is default value, defines the legacy mode, the `MARK` and `META` related actions and items operate only within NIC Tx and NIC Rx steering domains, no `MARK` and `META` information crosses the domain boundaries. The `MARK` item is 24 bits wide, the `META` item is 32 bits wide and match supported on egress only.

  – 1, this engages extensive metadata mode, the `MARK` and `META` related actions and items operate within all supported steering domains, including FDB, `MARK` and `META` information may cross the domain boundaries. The `MARK` item is 24 bits wide, the `META` item width depends on kernel and firmware configurations and might be 0, 16 or 32 bits. Within NIC Tx domain `META` data width is 32 bits for compatibility, the actual width of data transferred to the FDB domain depends on kernel configuration and may be vary. The actual supported width can be retrieved in runtime by series of rte_flow_validate() trials.

  – 2, this engages extensive metadata mode, the `MARK` and `META` related actions and items operate within all supported steering domains, including FDB, `MARK` and `META` information may cross the domain boundaries. The `META` item is 32 bits wide, the `MARK` item width depends on kernel and firmware configurations and might be 0, 16 or 24 bits. The actual supported width can be retrieved in runtime by series of rte_flow_validate() trials.

Mode	`MARK`	`META`	`META` Tx	FDB/Through
0	24 bits	32 bits	32 bits	no
1	24 bits	vary 0-32	32 bits	yes
2	vary 0-32	32 bits	32 bits	yes

  If there is no E-Switch configuration the `dv_xmeta_en` parameter is ignored and the device is configured to operate in legacy mode (0).

  Disabled by default (set to 0).

  The Direct Verbs/Rules (engaged with `dv_flow_en` = 1) supports all of the extensive metadata features. The legacy Verbs supports FLAG and MARK metadata actions over NIC Rx steering domain only.

- `dv_flow_en` parameter [int]

  A nonzero value enables the DV flow steering assuming it is supported by the driver (RDMA Core library version is rdma-core-24.0 or higher).

  Enabled by default if supported.

- `dv_esw_en` parameter [int]

  A nonzero value enables E-Switch using Direct Rules.

  Enabled by default if supported.

- `mr_ext_memseg_en` parameter [int]

  A nonzero value enables extending memseg when registering DMA memory. If enabled, the number of entries in MR (Memory Region) lookup table on datapath is minimized and it benefits performance. On the other hand, it worsens memory utilization because registered memory is pinned by kernel driver. Even if a page in the extended chunk is freed, that doesn't become reusable until the entire memory is freed.

  Enabled by default.

- `representor` parameter [list]

  This parameter can be used to instantiate DPDK Ethernet devices from existing port (or VF) representors configured on the device.

  It is a standard parameter whose format is described in *Ethernet Device Standard Device Arguments*.

  For instance, to probe port representors 0 through 2:

  ```
  representor=[0-2]
  ```

- `max_dump_files_num` parameter [int]

  The maximum number of files per PMD entity that may be created for debug information. The files will be created in /var/log directory or in current directory.

  set to 128 by default.

- `lro_timeout_usec` parameter [int]

  The maximum allowed duration of an LRO session, in micro-seconds. PMD will set the nearest value supported by HW, which is not bigger than the input `lro_timeout_usec` value. If this parameter is not specified, by default PMD will set the smallest value supported by HW.

- `hp_buf_log_sz` parameter [int]

  The total data buffer size of a hairpin queue (logarithmic form), in bytes. PMD will set the data buffer size to 2 ** `hp_buf_log_sz`, both for RX & TX. The capacity of the value is specified by the firmware and the initialization will get a failure if it is out of scope. The range of the value is from 11 to 19 right now, and the supported frame size of a single packet for hairpin is from 512B to 128KB. It might change if different firmware release is being used. By using a small value, it could reduce memory consumption but not work with a large frame. If the value is too large, the memory consumption will be high and some potential performance degradation will be introduced. By default, the PMD will set this value to 16, which means that 9KB jumbo frames will be supported.

Firmware configuration

Firmware features can be configured as key/value pairs.

The command to set a value is:

```
mlxconfig -d <device> set <key>=<value>
```

The command to query a value is:

```
mlxconfig -d <device> query | grep <key>
```

The device name for the command `mlxconfig` can be either the PCI address, or the mst device name found with:

```
mst status
```

Below are some firmware configurations listed.

- link type:

```
LINK_TYPE_P1
LINK_TYPE_P2
value: 1=Infiniband 2=Ethernet 3=VPI(auto-sense)
```

- enable SR-IOV:

```
SRIOV_EN=1
```

- maximum number of SR-IOV virtual functions:

```
NUM_OF_VFS=<max>
```

- enable DevX (required by Direct Rules and other features):

```
UCTX_EN=1
```

- aggressive CQE zipping:

```
CQE_COMPRESSION=1
```

- L3 VXLAN and VXLAN-GPE destination UDP port:

```
IP_OVER_VXLAN_EN=1
IP_OVER_VXLAN_PORT=<udp dport>
```

- enable VXLAN-GPE tunnel flow matching:

```
FLEX_PARSER_PROFILE_ENABLE=0
or
FLEX_PARSER_PROFILE_ENABLE=2
```

- enable IP-in-IP tunnel flow matching:

```
FLEX_PARSER_PROFILE_ENABLE=0
```

- enable MPLS flow matching:

```
FLEX_PARSER_PROFILE_ENABLE=1
```

- enable ICMP/ICMP6 code/type fields matching:

```
FLEX_PARSER_PROFILE_ENABLE=2
```

- enable Geneve flow matching:

```
FLEX_PARSER_PROFILE_ENABLE=0
or
FLEX_PARSER_PROFILE_ENABLE=1
```

- enable GTP flow matching:

```
FLEX_PARSER_PROFILE_ENABLE=3
```

9.34.6 Prerequisites

This driver relies on external libraries and kernel drivers for resources allocations and initialization. The following dependencies are not part of DPDK and must be installed separately:

- **libibverbs**

  User space Verbs framework used by librte_pmd_mlx5. This library provides a generic interface between the kernel and low-level user space drivers such as libmlx5.

  It allows slow and privileged operations (context initialization, hardware resources allocations) to be managed by the kernel and fast operations to never leave user space.

- **libmlx5**

  Low-level user space driver library for Mellanox ConnectX-4/ConnectX-5/ConnectX-6/BlueField devices, it is automatically loaded by libibverbs.

  This library basically implements send/receive calls to the hardware queues.

- **Kernel modules**

  They provide the kernel-side Verbs API and low level device drivers that manage actual hardware initialization and resources sharing with user space processes.

  Unlike most other PMDs, these modules must remain loaded and bound to their devices:

  - mlx5_core: hardware driver managing Mellanox ConnectX-4/ConnectX-5/ConnectX-6/BlueField devices and related Ethernet kernel network devices.
  - mlx5_ib: InifiniBand device driver.
  - ib_uverbs: user space driver for Verbs (entry point for libibverbs).

- **Firmware update**

  Mellanox OFED/EN releases include firmware updates for ConnectX-4/ConnectX-5/ConnectX-6/BlueField adapters.

  Because each release provides new features, these updates must be applied to match the kernel modules and libraries they come with.

Note: Both libraries are BSD and GPL licensed. Linux kernel modules are GPL licensed.

Installation

Either RDMA Core library with a recent enough Linux kernel release (recommended) or Mellanox OFED/EN, which provides compatibility with older releases.

RDMA Core with Linux Kernel

- Minimal kernel version : v4.14 or the most recent 4.14-rc (see Linux installation documentation)
- Minimal rdma-core version: v15+ commit 0c5f5765213a ("Merge pull request #227 from yishaih/tm") (see RDMA Core installation documentation)
- When building for i686 use:
  - rdma-core version 18.0 or above built with 32bit support.
  - Kernel version 4.14.41 or above.
- Starting with rdma-core v21, static libraries can be built:

```
cd build
CFLAGS=-fPIC cmake -DIN_PLACE=1 -DENABLE_STATIC=1 -GNinja ..
ninja
```

If rdma-core libraries are built but not installed, DPDK makefile can link them, thanks to these environment variables:

- `EXTRA_CFLAGS=-I/path/to/rdma-core/build/include`
- `EXTRA_LDFLAGS=-L/path/to/rdma-core/build/lib`
- `PKG_CONFIG_PATH=/path/to/rdma-core/build/lib/pkgconfig`

Mellanox OFED/EN

- Mellanox OFED version: **4.5** and above / Mellanox EN version: **4.5** and above
- firmware version:
  - ConnectX-4: **12.21.1000** and above.
  - ConnectX-4 Lx: **14.21.1000** and above.
  - ConnectX-5: **16.21.1000** and above.
  - ConnectX-5 Ex: **16.21.1000** and above.
  - ConnectX-6: **20.27.0090** and above.
  - ConnectX-6 Dx: **22.27.0090** and above.
  - BlueField: **18.25.1010** and above.

While these libraries and kernel modules are available on OpenFabrics Alliance's website and provided by package managers on most distributions, this PMD requires Ethernet extensions that may not be supported at the moment (this is a work in progress).

Mellanox OFED and Mellanox EN include the necessary support and should be used in the meantime. For DPDK, only libibverbs, libmlx5, mlnx-ofed-kernel packages and firmware updates are required from that distribution.

Note: Several versions of Mellanox OFED/EN are available. Installing the version this DPDK release was developed and tested against is strongly recommended. Please check the *prerequisites*.

9.34.7 Supported NICs

The following Mellanox device families are supported by the same mlx5 driver:

- ConnectX-4
- ConnectX-4 Lx
- ConnectX-5
- ConnectX-5 Ex
- ConnectX-6
- ConnectX-6 Dx
- BlueField

Below are detailed device names:

- Mellanox® ConnectX®-4 10G MCX4111A-XCAT (1x10G)
- Mellanox® ConnectX®-4 10G MCX412A-XCAT (2x10G)
- Mellanox® ConnectX®-4 25G MCX4111A-ACAT (1x25G)
- Mellanox® ConnectX®-4 25G MCX412A-ACAT (2x25G)
- Mellanox® ConnectX®-4 40G MCX413A-BCAT (1x40G)
- Mellanox® ConnectX®-4 40G MCX4131A-BCAT (1x40G)
- Mellanox® ConnectX®-4 40G MCX415A-BCAT (1x40G)
- Mellanox® ConnectX®-4 50G MCX413A-GCAT (1x50G)
- Mellanox® ConnectX®-4 50G MCX4131A-GCAT (1x50G)
- Mellanox® ConnectX®-4 50G MCX414A-BCAT (2x50G)
- Mellanox® ConnectX®-4 50G MCX415A-GCAT (1x50G)
- Mellanox® ConnectX®-4 50G MCX416A-BCAT (2x50G)
- Mellanox® ConnectX®-4 50G MCX416A-GCAT (2x50G)
- Mellanox® ConnectX®-4 50G MCX415A-CCAT (1x100G)
- Mellanox® ConnectX®-4 100G MCX416A-CCAT (2x100G)
- Mellanox® ConnectX®-4 Lx 10G MCX4111A-XCAT (1x10G)

- Mellanox® ConnectX®-4 Lx 10G MCX4121A-XCAT (2x10G)
- Mellanox® ConnectX®-4 Lx 25G MCX4111A-ACAT (1x25G)
- Mellanox® ConnectX®-4 Lx 25G MCX4121A-ACAT (2x25G)
- Mellanox® ConnectX®-4 Lx 40G MCX4131A-BCAT (1x40G)
- Mellanox® ConnectX®-5 100G MCX556A-ECAT (2x100G)
- Mellanox® ConnectX®-5 Ex EN 100G MCX516A-CDAT (2x100G)
- Mellanox® ConnectX®-6 200G MCX654106A-HCAT (2x200G)
- Mellanox® ConnectX®-6 Dx EN 100G MCX623106AN-CDAT (2x100G)
- Mellanox® ConnectX®-6 Dx EN 200G MCX623105AN-VDAT (1x200G)

9.34.8 Quick Start Guide on OFED/EN

1. Download latest Mellanox OFED/EN. For more info check the *prerequisites*.
2. Install the required libraries and kernel modules either by installing only the required set, or by installing the entire Mellanox OFED/EN:

```
./mlnxofedinstall --upstream-libs --dpdk
```

3. Verify the firmware is the correct one:

```
ibv_devinfo
```

4. Verify all ports links are set to Ethernet:

```
mlxconfig -d <mst device> query | grep LINK_TYPE
LINK_TYPE_P1                        ETH(2)
LINK_TYPE_P2                        ETH(2)
```

   Link types may have to be configured to Ethernet:

```
mlxconfig -d <mst device> set LINK_TYPE_P1/2=1/2/3

* LINK_TYPE_P1=<1|2|3> , 1=Infiniband 2=Ethernet 3=VPI(auto-sense)
```

   For hypervisors, verify SR-IOV is enabled on the NIC:

```
mlxconfig -d <mst device> query | grep SRIOV_EN
SRIOV_EN                            True(1)
```

   If needed, configure SR-IOV:

```
mlxconfig -d <mst device> set SRIOV_EN=1 NUM_OF_VFS=16
mlxfwreset -d <mst device> reset
```

5. Restart the driver:

```
/etc/init.d/openibd restart
```

   or:

```
service openibd restart
```

If link type was changed, firmware must be reset as well:

```
mlxfwreset -d <mst device> reset
```

For hypervisors, after reset write the sysfs number of virtual functions needed for the PF.

To dynamically instantiate a given number of virtual functions (VFs):

```
echo [num_vfs] > /sys/class/infiniband/mlx5_0/device/sriov_numvfs
```

6. Compile DPDK and you are ready to go. See instructions on *Development Kit Build System*

9.34.9 Enable switchdev mode

Switchdev mode is a mode in E-Switch, that binds between representor and VF. Representor is a port in DPDK that is connected to a VF in such a way that assuming there are no offload flows, each packet that is sent from the VF will be received by the corresponding representor. While each packet that is sent to a representor will be received by the VF. This is very useful in case of SRIOV mode, where the first packet that is sent by the VF will be received by the DPDK application which will decide if this flow should be offloaded to the E-Switch. After offloading the flow packet that the VF that are matching the flow will not be received any more by the DPDK application.

1. Enable SRIOV mode:

```
mlxconfig -d <mst device> set SRIOV_EN=true
```

2. Configure the max number of VFs:

```
mlxconfig -d <mst device> set NUM_OF_VFS=<num of vfs>
```

3. Reset the FW:

```
mlxfwreset -d <mst device> reset
```

3. Configure the actual number of VFs:

```
echo <num of vfs > /sys/class/net/<net device>/device/sriov_numvfs
```

4. Unbind the device (can be rebind after the switchdev mode):

```
echo -n "<device pci address" > /sys/bus/pci/drivers/mlx5_core/unbind
```

5. Enbale switchdev mode:

```
echo switchdev > /sys/class/net/<net device>/compat/devlink/mode
```

9.34.10 Performance tuning

1. Configure aggressive CQE Zipping for maximum performance:

```
mlxconfig -d <mst device> s CQE_COMPRESSION=1
```

   To set it back to the default CQE Zipping mode use:

```
mlxconfig -d <mst device> s CQE_COMPRESSION=0
```

2. In case of virtualization:
   - Make sure that hypervisor kernel is 3.16 or newer.
   - Configure boot with `iommu=pt`.
   - Use 1G huge pages.
   - Make sure to allocate a VM on huge pages.
   - Make sure to set CPU pinning.
3. Use the CPU near local NUMA node to which the PCIe adapter is connected, for better performance. For VMs, verify that the right CPU and NUMA node are pinned according to the above. Run:

```
lstopo-no-graphics
```

   to identify the NUMA node to which the PCIe adapter is connected.
4. If more than one adapter is used, and root complex capabilities allow to put both adapters on the same NUMA node without PCI bandwidth degradation, it is recommended to locate both adapters on the same NUMA node. This in order to forward packets from one to the other without NUMA performance penalty.
5. Disable pause frames:

```
ethtool -A <netdev> rx off tx off
```

6. Verify IO non-posted prefetch is disabled by default. This can be checked via the BIOS configuration. Please contact you server provider for more information about the settings.

Note: On some machines, depends on the machine integrator, it is beneficial to set the PCI max read request parameter to 1K. This can be done in the following way:

To query the read request size use:

```
setpci -s <NIC PCI address> 68.w
```

If the output is different than 3XXX, set it by:

```
setpci -s <NIC PCI address> 68.w=3XXX
```

The XXX can be different on different systems. Make sure to configure according to the setpci output.

---

7. To minimize overhead of searching Memory Regions:
   - '–socket-mem' is recommended to pin memory by predictable amount.

- Configure per-lcore cache when creating Mempools for packet buffer.
- Refrain from dynamically allocating/freeing memory in run-time.

9.34.11 Supported hardware offloads

Table 9.9: Minimal SW/HW versions for queue offloads

Offload	DPDK	Linux	rdma-core	OFED	firmware	hardware
common base	17.11	4.14	16	4.2-1	12.21.1000	ConnectX-4
checksums	17.11	4.14	16	4.2-1	12.21.1000	ConnectX-4
Rx timestamp	17.11	4.14	16	4.2-1	12.21.1000	ConnectX-4
TSO	17.11	4.14	16	4.2-1	12.21.1000	ConnectX-4
LRO	19.08	N/A	N/A	4.6-4	16.25.6406	ConnectX-5

Table 9.10: Minimal SW/HW versions for rte_flow offloads

Offload	with E-Switch	with NIC
Count	DPDK 19.05 OFED 4.6 rdma-core 24 ConnectX-5	DPDK 19.02 OFED 4.6 rdma-core 23 ConnectX-5
Drop	DPDK 19.05 OFED 4.6 rdma-core 24 ConnectX-5	DPDK 18.11 OFED 4.5 rdma-core 23 ConnectX-4
Queue / RSS	N/A	DPDK 18.11 OFED 4.5 rdma-core 23 ConnectX-4
Encapsulation (VXLAN / NVGRE / RAW)	DPDK 19.05 OFED 4.7-1 rdma-core 24 ConnectX-5	DPDK 19.02 OFED 4.6 rdma-core 23 ConnectX-5
Encapsulation GENEVE	DPDK 19.11 OFED 4.7-3 rdma-core 27 ConnectX-5	DPDK 19.11 OFED 4.7-3 rdma-core 27 ConnectX-5
Header rewrite (set_ipv4_src / set_ipv4_dst / set_ipv6_src / set_ipv6_dst / set_tp_src / set_tp_dst / dec_ttl / set_ttl / set_mac_src / set_mac_dst)	DPDK 19.05 OFED 4.7-1 rdma-core 24 ConnectX-5	DPDK 19.02 OFED 4.7-1 rdma-core 24 ConnectX-5
Header rewrite (set_dscp)	DPDK 20.02 OFED 5.0	DPDK 20.02 OFED 5.0

9.34.12 Notes for metadata

MARK and META items are interrelated with datapath - they might move from/to the applications in mbuf fields. Hence, zero value for these items has the special meaning - it means "no metadata are provided", not zero values are treated by applications and PMD as valid ones.

Moreover in the flow engine domain the value zero is acceptable to match and set, and we should allow to specify zero values as rte_flow parameters for the META and MARK items and actions. In the same time zero mask has no meaning and should be rejected on validation stage.

9.34.13 Notes for rte_flow

Flows are not cached in the driver. When stopping a device port, all the flows created on this port from the application will be flushed automatically in the background. After stopping the device port, all flows on this port become invalid and not represented in the system. All references to these flows held by the application should be discarded directly but neither destroyed nor flushed.

The application should re-create the flows as required after the port restart.

9.34.14 Notes for testpmd

Compared to librte_pmd_mlx4 that implements a single RSS configuration per port, librte_pmd_mlx5 supports per-protocol RSS configuration.

Since `testpmd` defaults to IP RSS mode and there is currently no command-line parameter to enable additional protocols (UDP and TCP as well as IP), the following commands must be entered from its CLI to get the same behavior as librte_pmd_mlx4:

```
> port stop all
> port config all rss all
> port start all
```

9.34.15 Usage example

This section demonstrates how to launch **testpmd** with Mellanox ConnectX-4/ConnectX-5/ConnectX-6/BlueField devices managed by librte_pmd_mlx5.

1. Load the kernel modules:

   ```
   modprobe -a ib_uverbs mlx5_core mlx5_ib
   ```

   Alternatively if MLNX_OFED/MLNX_EN is fully installed, the following script can be run:

   ```
   /etc/init.d/openibd restart
   ```

   Note: User space I/O kernel modules (uio and igb_uio) are not used and do not have to be loaded.

2. Make sure Ethernet interfaces are in working order and linked to kernel verbs. Related sysfs entries should be present:

```
ls -d /sys/class/net/*/device/infiniband_verbs/uverbs* | cut -d / -f 5
```

Example output:

```
eth30
eth31
eth32
eth33
```

3. Optionally, retrieve their PCI bus addresses for whitelisting:

```
{
    for intf in eth2 eth3 eth4 eth5;
    do
        (cd "/sys/class/net/${intf}/device/" && pwd -P);
    done;
} |
sed -n 's,.*/\(.*\),-w \1,p'
```

Example output:

```
-w 0000:05:00.1
-w 0000:06:00.0
-w 0000:06:00.1
-w 0000:05:00.0
```

4. Request huge pages:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages/nr_hugepages
```

5. Start testpmd with basic parameters:

```
testpmd -l 8-15 -n 4 -w 05:00.0 -w 05:00.1 -w 06:00.0 -w 06:00.1 -- --rxq=2 --txq=2 -i
```

Example output:

```
[...]
EAL: PCI device 0000:05:00.0 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_0" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fe
EAL: PCI device 0000:05:00.1 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_1" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:ff
EAL: PCI device 0000:06:00.0 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_2" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fa
EAL: PCI device 0000:06:00.1 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_3" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fb
Interactive-mode selected
Configuring Port 0 (socket 0)
```

(continues on next page)

(continued from previous page)

```
PMD: librte_pmd_mlx5: 0x8cba80: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8cba80: RX queues number update: 0 -> 2
Port 0: E4:1D:2D:E7:0C:FE
Configuring Port 1 (socket 0)
PMD: librte_pmd_mlx5: 0x8ccac8: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8ccac8: RX queues number update: 0 -> 2
Port 1: E4:1D:2D:E7:0C:FF
Configuring Port 2 (socket 0)
PMD: librte_pmd_mlx5: 0x8cdb10: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8cdb10: RX queues number update: 0 -> 2
Port 2: E4:1D:2D:E7:0C:FA
Configuring Port 3 (socket 0)
PMD: librte_pmd_mlx5: 0x8ceb58: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8ceb58: RX queues number update: 0 -> 2
Port 3: E4:1D:2D:E7:0C:FB
Checking link statuses...
Port 0 Link Up - speed 40000 Mbps - full-duplex
Port 1 Link Up - speed 40000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex
Port 3 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

9.34.16 How to dump flows

This section demonstrates how to dump flows. Currently, it's possible to dump all flows with assistance of external tools.

1. 2 ways to get flow raw file:

   - Using testpmd CLI:

```
testpmd> flow dump <port> <output_file>
```

   - call rte_flow_dev_dump api:

```
rte_flow_dev_dump(port, file, NULL);
```

2. Dump human-readable flows from raw file:

   Get flow parsing tool from: https://github.com/Mellanox/mlx_steering_dump

```
mlx_steering_dump.py -f <output_file>
```

9.35 MVNETA Poll Mode Driver

The MVNETA PMD (librte_pmd_mvneta) provides poll mode driver support for the Marvell NETA 1/2.5 Gbps adapter.

Detailed information about SoCs that use PPv2 can be obtained here:

- https://www.marvell.com/embedded-processors/armada-3700/

Note: Due to external dependencies, this driver is disabled by default. It must be enabled manually by setting relevant configuration option manually. Please refer to *Config File Options* section for further

details.

9.35.1 Features

Features of the MVNETA PMD are:

- Start/stop
- tx/rx_queue_setup
- tx/rx_burst
- Speed capabilities
- Jumbo frame
- MTU update
- Promiscuous mode
- Unicast MAC filter
- Link status
- CRC offload
- L3 checksum offload
- L4 checksum offload
- Packet type parsing
- Basic stats

9.35.2 Limitations

- Flushing vlans added for filtering is not possible due to MUSDK missing functionality. Current workaround is to reset board so that NETA has a chance to start in a sane state.

9.35.3 Prerequisites

- Custom Linux Kernel sources

```
git clone https://github.com/MarvellEmbeddedProcessors/linux-marvell.git -b linux-4.4.120-
↪armada-18.09
```

- MUSDK (Marvell User-Space SDK) sources

```
git clone https://github.com/MarvellEmbeddedProcessors/musdk-marvell.git -b musdk-armada-
↪18.09
```

MUSDK is a light-weight library that provides direct access to Marvell's NETA. Alternatively prebuilt MUSDK library can be requested from Marvell Extranet. Once approval has been granted, library can be found by typing `musdk` in the search box.

MUSDK must be configured with the following features:

```
--enable-pp2=no --enable-neta
```

- DPDK environment

  Follow the DPDK *Getting Started Guide for Linux* to setup DPDK environment.

9.35.4 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_MVNETA_PMD` (default `n`)

  Toggle compilation of the librte_pmd_mvneta driver.

Runtime options

The following `devargs` options can be enabled at runtime. They must be passed as part of EAL arguments.

- `iface` (mandatory, with no default value)

  The name of port (owned by MUSDK) that should be enabled in DPDK. This options can be repeated resulting in a list of ports to be enabled. For instance below will enable `eth0` and `eth1` ports.

```
./testpmd --vdev=net_mvneta,iface=eth0,iface=eth1 \
 -c 3 -- -i --p 3 -a
```

9.35.5 Building DPDK

Driver needs precompiled MUSDK library during compilation.

```
export CROSS_COMPILE=<toolchain>/bin/aarch64-linux-gnu-
./bootstrap
./configure --host=aarch64-linux-gnu --enable-pp2=no --enable-neta
make install
```

MUSDK will be installed to *usr/local* under current directory. For the detailed build instructions please consult `doc/musdk_get_started.txt`.

Before the DPDK build process the environmental variable `LIBMUSDK_PATH` with the path to the MUSDK installation directory needs to be exported.

```
export LIBMUSDK_PATH=<musdk>/usr/local
export CROSS=aarch64-linux-gnu-
make config T=arm64-armv8a-linux-gcc
sed -ri 's,(MVNETA_PMD=)n,\1y,' build/.config
make
```

9.35.6 Usage Example

MVNETA PMD requires extra out of tree kernel modules to function properly. *musdk_uio* and *mv_neta_uio* sources are part of the MUSDK. Please consult `doc/musdk_get_started.txt` for the detailed build instructions.

```
insmod musdk_uio.ko
insmod mv_neta_uio.ko
```

Additionally interfaces used by DPDK application need to be put up:

```
ip link set eth0 up
ip link set eth1 up
```

In order to run testpmd example application following command can be used:

```
./testpmd --vdev=net_mvneta,iface=eth0,iface=eth1 -c 3 -- \
  -i --p 3 -a --txd 256 --rxd 128 --rxq=1 --txq=1  --nb-cores=1
```

In order to run l2fwd example application following command can be used:

```
./l2fwd --vdev=net_mvneta,iface=eth0,iface=eth1 -c 3 -- -T 1 -p 3
```

9.36 MVPP2 Poll Mode Driver

The MVPP2 PMD (librte_pmd_mvpp2) provides poll mode driver support for the Marvell PPv2 (Packet Processor v2) 1/10 Gbps adapter.

Detailed information about SoCs that use PPv2 can be obtained here:

- https://www.marvell.com/embedded-processors/armada-70xx/
- https://www.marvell.com/embedded-processors/armada-80xx/

Note: Due to external dependencies, this driver is disabled by default. It must be enabled manually by setting relevant configuration option manually. Please refer to *Config File Options* section for further details.

9.36.1 Features

Features of the MVPP2 PMD are:

- Speed capabilities
- Link status
- Tx Queue start/stop
- MTU update
- Jumbo frame
- Promiscuous mode
- Allmulticast mode

- Unicast MAC filter
- Multicast MAC filter
- RSS hash
- VLAN filter
- CRC offload
- L3 checksum offload
- L4 checksum offload
- Packet type parsing
- Basic stats
- *Extended stats*
- RX flow control
- Scattered TX frames
- *QoS*
- *Flow API*
- *Traffic metering and policing*
- *Traffic Management API*

9.36.2 Limitations

- Number of lcores is limited to 9 by MUSDK internal design. If more lcores need to be allocated, locking will have to be considered. Number of available lcores can be changed via `MRVL_MUSDK_HIFS_RESERVED` define in `mrvl_ethdev.c` source file.
- Flushing vlans added for filtering is not possible due to MUSDK missing functionality. Current workaround is to reset board so that PPv2 has a chance to start in a sane state.
- MUSDK architecture does not support changing configuration in run time. All necessary configurations should be done before first dev_start().
- RX queue start/stop is not supported.
- Current implementation does not support replacement of buffers in the HW buffer pool at run time, so it is responsibility of the application to ensure that MTU does not exceed the configured buffer size.
- Configuring TX flow control currently is not supported.
- In current implementation, mechanism for acknowledging transmitted packets (`tx_done_cleanup`) is not supported.
- Running more than one DPDK-MUSDK application simultaneously is not supported.

9.36.3 Prerequisites

- Custom Linux Kernel sources

```
git clone https://github.com/MarvellEmbeddedProcessors/linux-marvell.git -b linux-4.4.120-
↪armada-18.09
```

- Out of tree *mvpp2x_sysfs* kernel module sources

```
git clone https://github.com/MarvellEmbeddedProcessors/mvpp2x-marvell.git -b mvpp2x-
↪armada-18.09
```

- MUSDK (Marvell User-Space SDK) sources

```
git clone https://github.com/MarvellEmbeddedProcessors/musdk-marvell.git -b musdk-armada-
↪18.09
```

  MUSDK is a light-weight library that provides direct access to Marvell's PPv2 (Packet Processor v2). Alternatively prebuilt MUSDK library can be requested from Marvell Extranet. Once approval has been granted, library can be found by typing `musdk` in the search box.

  To get better understanding of the library one can consult documentation available in the `doc` top level directory of the MUSDK sources.

- DPDK environment

  Follow the DPDK *Getting Started Guide for Linux* to setup DPDK environment.

9.36.4 Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_MVPP2_PMD` (default `n`)

  Toggle compilation of the librte mvpp2 driver.

  Note: When MVPP2 PMD is enabled `CONFIG_RTE_LIBRTE_MVNETA_PMD` must be disabled

9.36.5 Building DPDK

Driver needs precompiled MUSDK library during compilation.

```
export CROSS_COMPILE=<toolchain>/bin/aarch64-linux-gnu-
./bootstrap
./configure --host=aarch64-linux-gnu
make install
```

MUSDK will be installed to *usr/local* under current directory. For the detailed build instructions please consult `doc/musdk_get_started.txt`.

Before the DPDK build process the environmental variable `LIBMUSDK_PATH` with the path to the MUSDK installation directory needs to be exported.

For additional instructions regarding DPDK cross compilation please refer to *Cross compile DPDK for ARM64*.

```
export LIBMUSDK_PATH=<musdk>/usr/local
export CROSS=<toolchain>/bin/aarch64-linux-gnu-
export RTE_KERNELDIR=<kernel-dir>
export RTE_TARGET=arm64-armv8a-linux-gcc

make config T=arm64-armv8a-linux-gcc
sed -i "s/MVNETA_PMD=y/MVNETA_PMD=n/" build/.config
sed -i "s/MVPP2_PMD=n/MVPP2_PMD=y/" build/.config
make
```

9.36.6 Usage Example

MVPP2 PMD requires extra out of tree kernel modules to function properly. *musdk_cma* sources are part of the MUSDK. Please consult `doc/musdk_get_started.txt` for the detailed build instructions. For *mvpp2x_sysfs* please consult `Documentation/pp22_sysfs.txt` for the detailed build instructions.

```
insmod musdk_cma.ko
insmod mvpp2x_sysfs.ko
```

Additionally interfaces used by DPDK application need to be put up:

```
ip link set eth0 up
ip link set eth2 up
```

In order to run testpmd example application following command can be used:

```
./testpmd --vdev=eth_mvpp2,iface=eth0,iface=eth2 -c 7 -- \
  --burst=128 --txd=2048 --rxd=1024 --rxq=2 --txq=2  --nb-cores=2 \
  -i -a --rss-udp
```

9.36.7 Extended stats

MVPP2 PMD supports the following extended statistics:

- `rx_bytes`: number of RX bytes
- `rx_packets`: number of RX packets
- `rx_unicast_packets`: number of RX unicast packets
- `rx_errors`: number of RX MAC errors
- `rx_fullq_dropped`: number of RX packets dropped due to full RX queue
- `rx_bm_dropped`: number of RX packets dropped due to no available buffers in the HW pool
- `rx_early_dropped`: number of RX packets that were early dropped
- `rx_fifo_dropped`: number of RX packets dropped due to RX fifo overrun
- `rx_cls_dropped`: number of RX packets dropped by classifier
- `tx_bytes`: number of TX bytes
- `tx_packets`: number of TX packets

- `tx_unicast_packets`: number of TX unicast packets
- `tx_errors`: number of TX MAC errors

9.36.8 QoS Configuration

QoS configuration is done through external configuration file. Path to the file must be given as *cfg* in driver's vdev parameter list.

Configuration syntax

```
[policer <policer_id>]
token_unit = <token_unit>
color = <color_mode>
cir = <cir>
ebs = <ebs>
cbs = <cbs>

[port <portnum> default]
default_tc = <default_tc>
mapping_priority = <mapping_priority>

rate_limit_enable = <rate_limit_enable>
rate_limit = <rate_limit>
burst_size = <burst_size>

default_policer = <policer_id>

[port <portnum> tc <traffic_class>]
rxq = <rx_queue_list>
pcp = <pcp_list>
dscp = <dscp_list>
default_color = <default_color>

[port <portnum> tc <traffic_class>]
rxq = <rx_queue_list>
pcp = <pcp_list>
dscp = <dscp_list>

[port <portnum> txq <txqnum>]
sched_mode = <sched_mode>
wrr_weight = <wrr_weight>

rate_limit_enable = <rate_limit_enable>
rate_limit = <rate_limit>
burst_size = <burst_size>
```

Where:

- `<portnum>`: DPDK Port number (0..n).
- `<default_tc>`: Default traffic class (e.g. 0)
- `<mapping_priority>`: QoS priority for mapping (*ip*, *vlan*, *ip/vlan* or *vlan/ip*).
- `<traffic_class>`: Traffic Class to be configured.
- `<rx_queue_list>`: List of DPDK RX queues (e.g. 0 1 3-4)
- `<pcp_list>`: List of PCP values to handle in particular TC (e.g. 0 1 3-4 7).

- <dscp_list>: List of DSCP values to handle in particular TC (e.g. 0-12 32-48 63).
- <default_policer>: Id of the policer configuration section to be used as default.
- <policer_id>: Id of the policer configuration section (0..31).
- <token_unit>: Policer token unit (*bytes* or *packets*).
- <color_mode>: Policer color mode (*aware* or *blind*).
- <cir>: Committed information rate in unit of kilo bits per second (data rate) or packets per second.
- <cbs>: Committed burst size in unit of kilo bytes or number of packets.
- <ebs>: Excess burst size in unit of kilo bytes or number of packets.
- <default_color>: Default color for specific tc.
- <rate_limit_enable>: Enables per port or per txq rate limiting (*0*/*1* to disable/enable).
- <rate_limit>: Committed information rate, in kilo bits per second.
- <burst_size>: Committed burst size, in kilo bytes.
- <sched_mode>: Egress scheduler mode (*wrr* or *sp*).
- <wrr_weight>: Txq weight.

Setting PCP/DSCP values for the default TC is not required. All PCP/DSCP values not assigned explicitly to particular TC will be handled by the default TC.

Configuration file example

```
[policer 0]
token_unit = bytes
color = blind
cir = 100000
ebs = 64
cbs = 64

[port 0 default]
default_tc = 0
mapping_priority = ip

rate_limit_enable = 1
rate_limit = 1000
burst_size = 2000

[port 0 tc 0]
rxq = 0 1

[port 0 txq 0]
sched_mode = wrr
wrr_weight = 10

[port 0 txq 1]
sched_mode = wrr
wrr_weight = 100

[port 0 txq 2]
sched_mode = sp
```

(continues on next page)

(continued from previous page)

```
[port 0 tc 1]
rxq = 2
pcp = 5 6 7
dscp = 26-38

[port 1 default]
default_tc = 0
mapping_priority = vlan/ip

default_policer = 0

[port 1 tc 0]
rxq = 0
dscp = 10

[port 1 tc 1]
rxq = 1
dscp = 11-20

[port 1 tc 2]
rxq = 2
dscp = 30

[port 1 txq 0]
rate_limit_enable = 1
rate_limit = 10000
burst_size = 2000
```

Usage example

```
./testpmd --vdev=eth_mvpp2,iface=eth0,iface=eth2,cfg=/home/user/mrvl.conf \
  -c 7 -- -i -a --disable-hw-vlan-strip --rxq=3 --txq=3
```

9.36.9 Flow API

PPv2 offers packet classification capabilities via classifier engine which can be configured via generic flow API offered by DPDK.

The *Flow isolated mode* is supported.

For an additional description please refer to DPDK *Generic flow API (rte_flow)*.

Supported flow actions

Following flow action items are supported by the driver:

- DROP
- QUEUE

Supported flow items

Following flow items and their respective fields are supported by the driver:

- ETH
  - source MAC
  - destination MAC
  - ethertype
- VLAN
  - PCP
  - VID
- IPV4
  - DSCP
  - protocol
  - source address
  - destination address
- IPV6
  - flow label
  - next header
  - source address
  - destination address
- UDP
  - source port
  - destination port
- TCP
  - source port
  - destination port

Classifier match engine

Classifier has an internal match engine which can be configured to operate in either exact or maskable mode.

Mode is selected upon creation of the first unique flow rule as follows:

- maskable, if key size is up to 8 bytes.
- exact, otherwise, i.e for keys bigger than 8 bytes.

Where the key size equals the number of bytes of all fields specified in the flow items.

Table 9.11: Examples of key size calculation

Flow pattern	Key size in bytes	Used engine
ETH (destination MAC) / VLAN (VID)	6 + 2 = 8	Maskable
VLAN (VID) / IPV4 (source address)	2 + 4 = 6	Maskable
TCP (source port, destination port)	2 + 2 = 4	Maskable
VLAN (priority) / IPV4 (source address)	1 + 4 = 5	Maskable
IPV4 (destination address) / UDP (source port, destination port)	6 + 2 + 2 = 10	Exact
VLAN (VID) / IPV6 (flow label, destination address)	2 + 3 + 16 = 21	Exact
IPV4 (DSCP, source address, destination address)	1 + 4 + 4 = 9	Exact
IPV6 (flow label, source address, destination address)	3 + 16 + 16 = 35	Exact

From the user perspective maskable mode means that masks specified via flow rules are respected. In case of exact match mode, masks which do not provide exact matching (all bits masked) are ignored.

If the flow matches more than one classifier rule the first (with the lowest index) matched takes precedence.

Flow rules usage example

Before proceeding run testpmd user application:

```
./testpmd --vdev=eth_mvpp2,iface=eth0,iface=eth2 -c 3 -- -i --p 3 -a --disable-hw-vlan-strip
```

Example #1

```
testpmd> flow create 0 ingress pattern eth src is 10:11:12:13:14:15 / end actions drop / end
```

In this case key size is 6 bytes thus maskable type is selected. Testpmd will set mask to ff:ff:ff:ff:ff:ff i.e traffic explicitly matching above rule will be dropped.

Example #2

```
testpmd> flow create 0 ingress pattern ipv4 src spec 10.10.10.0 src mask 255.255.255.0 / tcp␣
↪src spec 0x10 src mask 0x10 / end action drop / end
```

In this case key size is 8 bytes thus maskable type is selected. Flows which have IPv4 source addresses ranging from 10.10.10.0 to 10.10.10.255 and tcp source port set to 16 will be dropped.

Example #3

```
testpmd> flow create 0 ingress pattern vlan vid spec 0x10 vid mask 0x10 / ipv4 src spec 10.10.
↪1.1 src mask 255.255.0.0 dst spec 11.11.11.1 dst mask 255.255.255.0 / end actions drop / end
```

In this case key size is 10 bytes thus exact type is selected. Even though each item has partial mask set, masks will be ignored. As a result only flows with VID set to 16 and IPv4 source and destination addresses set to 10.10.1.1 and 11.11.11.1 respectively will be dropped.

Limitations

Following limitations need to be taken into account while creating flow rules:

- For IPv4 exact match type the key size must be up to 12 bytes.
- For IPv6 exact match type the key size must be up to 36 bytes.
- Following fields cannot be partially masked (all masks are treated as if they were exact):
  - ETH: ethertype
  - VLAN: PCP, VID
  - IPv4: protocol
  - IPv6: next header
  - TCP/UDP: source port, destination port
- Only one classifier table can be created thus all rules in the table have to match table format. Table format is set during creation of the first unique flow rule.
- Up to 5 fields can be specified per flow rule.
- Up to 20 flow rules can be added.

For additional information about classifier please consult `doc/musdk_cls_user_guide.txt`.

9.36.10 Traffic metering and policing

MVPP2 PMD supports DPDK traffic metering and policing that allows the following:

1. Meter ingress traffic.
2. Do policing.
3. Gather statistics.

For an additional description please refer to DPDK *Traffic Metering and Policing API*.

The policer objects defined by this feature can work with the default policer defined via config file as described in *QoS Support*.

Limitations

The following capabilities are not supported:

- MTR object meter DSCP table update
- MTR object policer action update
- MTR object enabled statistics

Usage example

1. Run testpmd user app:

```
./testpmd --vdev=eth_mvpp2,iface=eth0,iface=eth2 -c 6 -- -i -p 3 -a --txd 1024 --rxd 1024
```

2. Create meter profile:

```
testpmd> add port meter profile 0 0 srtcm_rfc2697 2000 256 256
```

3. Create meter:

```
testpmd> create port meter 0 0 0 yes d d d 0 1 0
```

4. Create flow rule witch meter attached:

```
testpmd> flow create 0 ingress pattern ipv4 src is 10.10.10.1 / end actions meter mtr_id␣
↪0 / end
```

For a detailed usage description please refer to "Traffic Metering and Policing" section in DPDK *Testpmd Runtime Functions*.

9.36.11 Traffic Management API

MVPP2 PMD supports generic DPDK Traffic Management API which allows to configure the following features:

1. Hierarchical scheduling
2. Traffic shaping
3. Congestion management
4. Packet marking

Internally TM is represented by a hierarchy (tree) of nodes. Node which has a parent is called a leaf whereas node without parent is called a non-leaf (root). MVPP2 PMD supports two level hierarchy where level 0 represents ports and level 1 represents tx queues of a given port.

Nodes hold following types of settings:

- for egress scheduler configuration: weight
- for egress rate limiter: private shaper
- bitmask indicating which statistics counters will be read

Hierarchy is always constructed from the top, i.e first a root node is added then some number of leaf nodes. Number of leaf nodes cannot exceed number of configured tx queues.

After hierarchy is complete it can be committed.

For an additional description please refer to DPDK *Traffic Management API*.

Limitations

The following capabilities are not supported:

- Traffic manager WRED profile and WRED context
- Traffic manager shared shaper update
- Traffic manager packet marking
- Maximum number of levels in hierarchy is 2
- Currently dynamic change of a hierarchy is not supported

Usage example

For a detailed usage description please refer to "Traffic Management" section in DPDK *Testpmd Runtime Functions*.

1. Run testpmd as follows:

```
./testpmd --vdev=net_mrvl,iface=eth0,iface=eth2,cfg=./qos_config -c 7 -- \
-i -p 3 --disable-hw-vlan-strip --rxq 3 --txq 3 --txd 1024 --rxd 1024
```

2. Stop all ports:

```
testpmd> port stop all
```

3. Add shaper profile:

```
testpmd> add port tm node shaper profile 0 0 900000 70000 0
```

   Parameters have following meaning:

```
0       - Id of a port.
0       - Id of a new shaper profile.
900000  - Shaper rate in bytes/s.
70000   - Bucket size in bytes.
0       - Packet length adjustment - ignored.
```

4. Add non-leaf node for port 0:

```
testpmd> add port tm nonleaf node 0 3 -1 0 0 0 0 0 1 3 0
```

   Parameters have following meaning:

```
 0  - Id of a port
 3  - Id of a new node.
-1  - Indicate that root does not have a parent.
 0  - Priority of the node.
```

(continues on next page)

(continued from previous page)

```
 0  - Weight of the node.
 0  - Id of a level. Since this is a root 0 is passed.
 0  - Id of the shaper profile.
 0  - Number of SP priorities.
 3  - Enable statistics for both number of transmitted packets and bytes.
 0  - Number of shared shapers.
```

5. Add leaf node for tx queue 0:

```
testpmd> add port tm leaf node 0 0 3 0 30 1 -1 0 0 1 0
```

Parameters have following meaning:

```
 0  - Id of a port.
 0  - Id of a new node.
 3  - Id of the parent node.
 0  - Priority of a node.
 30 - WRR weight.
 1  - Id of a level. Since this is a leaf node 1 is passed.
-1  - Id of a shaper. -1 indicates that shaper is not attached.
 0  - Congestion management is not supported.
 0  - Congestion management is not supported.
 1  - Enable statistics counter for number of transmitted packets.
 0  - Number of shared shapers.
```

6. Add leaf node for tx queue 1:

```
testpmd> add port tm leaf node 0 1 3 0 60 1 -1 0 0 1 0
```

Parameters have following meaning:

```
 0  - Id of a port.
 1  - Id of a new node.
 3  - Id of the parent node.
 0  - Priority of a node.
 60 - WRR weight.
 1  - Id of a level. Since this is a leaf node 1 is passed.
-1  - Id of a shaper. -1 indicates that shaper is not attached.
 0  - Congestion management is not supported.
 0  - Congestion management is not supported.
 1  - Enable statistics counter for number of transmitted packets.
 0  - Number of shared shapers.
```

7. Add leaf node for tx queue 2:

```
testpmd> add port tm leaf node 0 2 3 0 99 1 -1 0 0 1 0
```

Parameters have following meaning:

```
 0  - Id of a port.
 2  - Id of a new node.
 3  - Id of the parent node.
 0  - Priority of a node.
 99 - WRR weight.
 1  - Id of a level. Since this is a leaf node 1 is passed.
-1  - Id of a shaper. -1 indicates that shaper is not attached.
 0  - Congestion management is not supported.
 0  - Congestion management is not supported.
```

(continues on next page)

(continued from previous page)

```
1  - Enable statistics counter for number of transmitted packets.
0  - Number of shared shapers.
```

8. Commit hierarchy:

```
testpmd> port tm hierarchy commit 0 no
```

Parameters have following meaning:

```
0  - Id of a port.
no - Do not flush TM hierarchy if commit fails.
```

9. Start all ports

```
testpmd> port start all
```

10. Enable forwarding

```
testpmd> start
```

9.37 Netvsc poll mode driver

The Netvsc Poll Mode driver (PMD) provides support for the paravirtualized network device for Microsoft Hyper-V. It can be used with Window Server 2008/2012/2016, Windows 10. The device offers multi-queue support (if kernel and host support it), checksum and segmentation offloads.

9.37.1 Features and Limitations of Hyper-V PMD

In this release, the hyper PMD driver provides the basic functionality of packet reception and transmission.

- It supports merge-able buffers per packet when receiving packets and scattered buffer per packet when transmitting packets. The packet size supported is from 64 to 65536.
- The PMD supports multicast packets and promiscuous mode subject to restrictions on the host. In order to this to work, the guest network configuration on Hyper-V must be configured to allow MAC address spoofing.
- The device has only a single MAC address. Hyper-V driver does not support MAC or VLAN filtering because the Hyper-V host does not support it.
- VLAN tags are always stripped and presented in mbuf tci field.
- The Hyper-V driver does not use or support interrupts. Link state change callback is done via change events in the packet ring.
- The maximum number of queues is limited by the host (currently 64). When used with 4.16 kernel only a single queue is available.
- This driver supports SR-IOV network acceleration. If SR-IOV is enabled then the driver will transparently manage the interface, and send and receive packets using the VF path. The VDEV_NETVSC and FAILSAFE drivers are *not* used when using netvsc PMD.

9.37.2 Installation

The Netvsc PMD is a standalone driver, similar to virtio and vmxnet3. Using Netvsc PMD requires that the associated VMBUS device be bound to the userspace I/O device driver for Hyper-V (uio_hv_generic). By default, all netvsc devices will be bound to the Linux kernel driver; in order to use netvsc PMD the device must first be overridden.

The first step is to identify the network device to override. VMBUS uses Universal Unique Identifiers (UUID) to identify devices on the bus similar to how PCI uses Domain:Bus:Function. The UUID associated with a Linux kernel network device can be determined by looking at the sysfs information. To find the UUID for eth1 and store it in a shell variable:

```
DEV_UUID=$(basename $(readlink /sys/class/net/eth1/device))
```

There are several possible ways to assign the uio device driver for a device. The easiest way (but only on 4.18 or later) is to use the driverctl Device Driver control utility to override the normal kernel device.

```
driverctl -b vmbus set-override $DEV_UUID uio_hv_generic
```

Any settings done with driverctl are by default persistent and will be reapplied on reboot.

On older kernels, the same effect can be had by manual sysfs bind and unbind operations:

```
NET_UUID="f8615163-df3e-46c5-913f-f2d2f965ed0e"
modprobe uio_hv_generic
echo $NET_UUID > /sys/bus/vmbus/drivers/uio_hv_generic/new_id
echo $DEV_UUID > /sys/bus/vmbus/drivers/hv_netvsc/unbind
echo $DEV_UUID > /sys/bus/vmbus/drivers/uio_hv_generic/bind
```

Note: The dpdk-devbind.py script can not be used since it only handles PCI devices.

9.37.3 Prerequisites

The following prerequisites apply:

- Linux kernel support for UIO on vmbus is done with the uio_hv_generic driver. Full support of multiple queues requires the 4.17 kernel. It is possible to use the netvsc PMD with 4.16 kernel but it is limited to a single queue.

9.37.4 Netvsc PMD arguments

The user can specify below argument in devargs.

1. `latency`:

   A netvsc device uses a mailbox page to indicate to the host that there is something in the transmit queue. The host scans this page at a periodic interval. This parameter allows adjusting the value that is used by the host. Smaller values improve transmit latency, and larger values save CPU cycles. This parameter is in microseconds. If the value is too large or too small it will be ignored by the host. (Default: 50)

9.38 NFB poll mode driver library

The NFB poll mode driver library implements support for the Netcope FPGA Boards (**NFB-40G2, NFB-100G2, NFB-200G2QL**) and Silicom **FB2CGG3** card, FPGA-based programmable NICs. The NFB PMD uses interface provided by the libnfb library to communicate with these cards over the nfb layer.

More information about the NFB cards and used technology (Netcope Development Kit) can be found on the Netcope Technologies website.

Note: This driver has external dependencies. Therefore it is disabled in default configuration files. It can be enabled by setting `CONFIG_RTE_LIBRTE_NFB_PMD=y` and recompiling.

Note: Currently the driver is supported only on x86_64 architectures. Only x86_64 versions of the external libraries are provided.

9.38.1 Prerequisites

This PMD requires kernel modules which are responsible for initialization and allocation of resources needed for nfb layer function. Communication between PMD and kernel modules is mediated by libnfb library. These kernel modules and library are not part of DPDK and must be installed separately:

- **libnfb library**

  The library provides API for initialization of nfb transfers, receiving and transmitting data segments.

- **Kernel modules**

  - nfb

  Kernel modules manage initialization of hardware, allocation and sharing of resources for user space applications.

Dependencies can be found here: Netcope common.

Versions of the packages

The minimum version of the provided packages:

- for DPDK from 19.05

9.38.2 Configuration

These configuration options can be modified before compilation in the `.config` file:

- `CONFIG_RTE_LIBRTE_NFB_PMD` default value: **n**

  Value **y** enables compilation of nfb PMD.

Timestamps

The PMD supports hardware timestamps of frame receipt on physical network interface. In order to use the timestamps, the hardware timestamping unit must be enabled (follow the documentation of the NFB products) and the device argument *timestamp=1* must be used.

```
$RTE_TARGET/app/testpmd -w b3:00.0,timestamp=1 <other EAL params> -- <testpmd params>
```

When the timestamps are enabled with the *devarg*, a timestamp validity flag is set in the MBUFs containing received frames and timestamp is inserted into the *rte_mbuf* struct.

The timestamp is an *uint64_t* field. Its lower 32 bits represent *seconds* portion of the timestamp (number of seconds elapsed since 1.1.1970 00:00:00 UTC) and its higher 32 bits represent *nanosecond* portion of the timestamp (number of nanoseconds elapsed since the beginning of the second in the *seconds* portion.

9.38.3 Using the NFB PMD

Kernel modules have to be loaded before running the DPDK application.

9.38.4 NFB card architecture

The NFB cards are multi-port multi-queue cards, where (generally) data from any Ethernet port may be sent to any queue. They are represented in DPDK as a single port.

NFB-200G2QL card employs an add-on cable which allows to connect it to two physical PCI-E slots at the same time (see the diagram below). This is done to allow 200 Gbps of traffic to be transferred through the PCI-E bus (note that a single PCI-E 3.0 x16 slot provides only 125 Gbps theoretical throughput).

Although each slot may be connected to a different CPU and therefore to a different NUMA node, the card is represented as a single port in DPDK. To work with data from the individual queues on the right NUMA node, connection of NUMA nodes on first and last queue (each NUMA node has half of the queues) need to be checked.

Fig. 9.7: NFB-200G2QL high-level diagram

9.38.5 Limitations

Driver is usable only on Linux architecture, namely on CentOS.

Since a card is always represented as a single port, but can be connected to two NUMA nodes, there is need for manual check where master/slave is connected.

9.38.6 Example of usage

Read packets from 0. and 1. receive queue and write them to 0. and 1. transmit queue:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 2 \
-- --port-topology=chained --rxq=2 --txq=2 --nb-cores=2 -i -a
```

Example output:

```
[...]
EAL: PCI device 0000:06:00.0 on NUMA socket -1
EAL:   probe driver: 1b26:c1c1 net_nfb
PMD: Initializing NFB device (0000:06:00.0)
PMD: Available DMA queues RX: 8 TX: 8
PMD: NFB device (0000:06:00.0) successfully initialized
Interactive-mode selected
Auto-start selected
Configuring Port 0 (socket 0)
Port 0: 00:11:17:00:00:00
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
Start automatic packet forwarding
  io packet forwarding - CRC stripping disabled - packets/burst=32
  nb forwarding cores=2 - nb forwarding ports=1
  RX queues=2 - RX desc=128 - RX free threshold=0
  RX threshold registers: pthresh=0 hthresh=0 wthresh=0
  TX queues=2 - TX desc=512 - TX free threshold=0
  TX threshold registers: pthresh=0 hthresh=0 wthresh=0
  TX RS bit threshold=0 - TXQ flags=0x0
testpmd>
```

9.39 NFP poll mode driver library

Netronome's sixth generation of flow processors pack 216 programmable cores and over 100 hardware accelerators that uniquely combine packet, flow, security and content processing in a single device that scales up to 400-Gb/s.

This document explains how to use DPDK with the Netronome Poll Mode Driver (PMD) supporting Netronome's Network Flow Processor 6xxx (NFP-6xxx) and Netronome's Flow Processor 4xxx (NFP-4xxx).

NFP is a SRIOV capable device and the PMD driver supports the physical function (PF) and the virtual functions (VFs).

9.39.1 Dependencies

Before using the Netronome's DPDK PMD some NFP configuration, which is not related to DPDK, is required. The system requires installation of **Netronome's BSP (Board Support Package)** along with a specific NFP firmware application. Netronome's NSP ABI version should be 0.20 or higher.

If you have a NFP device you should already have the code and documentation for this configuration. Contact **support@netronome.com** to obtain the latest available firmware.

The NFP Linux netdev kernel driver for VFs has been a part of the vanilla kernel since kernel version 4.5, and support for the PF since kernel version 4.11. Support for older kernels can be obtained on Github at **https://github.com/Netronome/nfp-drv-kmods** along with the build instructions.

NFP PMD needs to be used along with UIO `igb_uio` or VFIO (`vfio-pci`) Linux kernel driver.

9.39.2 Building the software

Netronome's PMD code is provided in the **drivers/net/nfp** directory. Although NFP PMD has Netronome´s BSP dependencies, it is possible to compile it along with other DPDK PMDs even if no BSP was installed previously. Of course, a DPDK app will require such a BSP installed for using the NFP PMD, along with a specific NFP firmware application.

Default PMD configuration is at the **common_linux configuration** file:

- **CONFIG_RTE_LIBRTE_NFP_PMD=y**

Once the DPDK is built all the DPDK apps and examples include support for the NFP PMD.

9.39.3 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

9.39.4 Using the PF

NFP PMD supports using the NFP PF as another DPDK port, but it does not have any functionality for controlling VFs. In fact, it is not possible to use the PMD with the VFs if the PF is being used by DPDK, that is, with the NFP PF bound to `igb_uio` or `vfio-pci` kernel drivers. Future DPDK versions will have a PMD able to work with the PF and VFs at the same time and with the PF implementing VF management along with other PF-only functionalities/offloads.

The PMD PF has extra work to do which will delay the DPDK app initialization like uploading the firmware and configure the Link state properly when starting or stopping a PF port. Since DPDK 18.05 the firmware upload happens when a PF is initialized, which was not always true with older DPDK versions.

Depending on the Netronome product installed in the system, firmware files should be available under `/lib/firmware/netronome`. DPDK PMD supporting the PF looks for a firmware file in this order:

1) First try to find a firmware image specific for this device using the NFP serial number:

   serial-00-15-4d-12-20-65-10-ff.nffw

2) Then try the PCI name:

   pci-0000:04:00.0.nffw

3) Finally try the card type and media:

   nic_AMDA0099-0001_2x25.nffw

Netronome's software packages install firmware files under `/lib/firmware/netronome` to support all the Netronome's SmartNICs and different firmware applications. This is usually done using file names based on SmartNIC type and media and with a directory per firmware application. Options 1 and 2 for firmware filenames allow more than one SmartNIC, same type of SmartNIC or different ones, and to upload a different firmware to each SmartNIC.

9.39.5 PF multiport support

Some NFP cards support several physical ports with just one single PCI device. The DPDK core is designed with a 1:1 relationship between PCI devices and DPDK ports, so NFP PMD PF support requires handling the multiport case specifically. During NFP PF initialization, the PMD will extract the information about the number of PF ports from the firmware and will create as many DPDK ports as needed.

Because the unusual relationship between a single PCI device and several DPDK ports, there are some limitations when using more than one PF DPDK port: there is no support for RX interrupts and it is not possible either to use those PF ports with the device hotplug functionality.

9.39.6 PF multiprocess support

Due to how the driver needs to access the NFP through a CPP interface, which implies to use specific registers inside the chip, the number of secondary processes with PF ports is limited to only one.

This limitation will be solved in future versions but having basic multiprocess support is important for allowing development and debugging through the PF using a secondary process which will create a CPP bridge for user space tools accessing the NFP.

9.39.7 System configuration

1. **Enable SR-IOV on the NFP device:** The current NFP PMD supports the PF and the VFs on a NFP device. However, it is not possible to work with both at the same time because the VFs require the PF being bound to the NFP PF Linux netdev driver. Make sure you are working with a kernel with NFP PF support or get the drivers from the above Github repository and follow the instructions for building and installing it.

   VFs need to be enabled before they can be used with the PMD. Before enabling the VFs it is useful to obtain information about the current NFP PCI device detected by the system:

   ```
   lspci -d19ee:
   ```

   Now, for example, configure two virtual functions on a NFP-6xxx device whose PCI system identity is "0000:03:00.0":

   ```
   echo 2 > /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs
   ```

   The result of this command may be shown using lspci again:

   ```
   lspci -d19ee: -k
   ```

   Two new PCI devices should appear in the output of the above command. The -k option shows the device driver, if any, that devices are bound to. Depending on the modules loaded at this point the new PCI devices may be bound to nfp_netvf driver.

9.40 NULL Poll Mode Driver

NULL PMD is a simple virtual driver mainly for testing. It always returns success for all packets for Rx/Tx.

On Rx it returns requested number of empty packets (all zero). On Tx it just frees all sent packets.

9.40.1 Usage

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev net_null0 --vdev net_null1 -- -i
```

9.40.2 Runtime Config Options

- `copy` [optional, default disabled]

  It copies data of the packet before Rx/Tx. For Rx it uses another empty dummy mbuf for this.

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev "net_null0,copy=1" -- -i
```

- `size` [optional, default=64 bytes]

  Custom packet length value to use.r If `copy` is enabled, this is the length of copy operation.

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 --vdev "net_null0,size=256" -- -i
```

- `no-rx` [optional, default disabled]

  Makes PMD more like `/dev/null`. On Rx no packets received, on Tx all packets are freed. This option can't co-exist with `copy` option.

9.41 OCTEON TX Poll Mode driver

The OCTEON TX ETHDEV PMD (**librte_pmd_octeontx**) provides poll mode ethdev driver support for the inbuilt network device found in the **Cavium OCTEON TX** SoC family as well as their virtual functions (VF) in SR-IOV context.

More information can be found at Cavium, Inc Official Website.

9.41.1 Features

Features of the OCTEON TX Ethdev PMD are:

- Packet type information
- Promiscuous mode
- Port hardware statistics
- Jumbo frames
- Scatter-Gather IO support

- Link state information
- MAC/VLAN filtering
- MTU update
- SR-IOV VF
- Multiple queues for TX
- Lock-free Tx queue
- HW offloaded *ethdev Rx queue* to *eventdev event queue* packet injection

9.41.2 Supported OCTEON TX SoCs

- CN83xx

9.41.3 Unsupported features

The features supported by the device and not yet supported by this PMD include:

- Receive Side Scaling (RSS)
- Scattered and gather for TX and RX
- Ingress classification support
- Egress hierarchical scheduling, traffic shaping, and marking

9.41.4 Prerequisites

See *OCTEON TX Board Support Package* for setup information.

9.41.5 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_OCTEONTX_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_octeontx` driver.

Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

To compile the OCTEON TX PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-thunderx-linux-gcc install
```

1. Running testpmd:

   Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

   Example output:

```
./arm64-thunderx-linux-gcc/app/testpmd -c 700 \
          --base-virtaddr=0x100000000000 \
          --mbuf-pool-ops-name="octeontx_fpavf" \
          --vdev='event_octeontx' \
          --vdev='eth_octeontx,nr_port=2' \
          -- --rxq=1 --txq=1 --nb-core=2 \
          --total-num-mbufs=16384 -i
.....
EAL: Detected 24 lcore(s)
EAL: Probing VFIO support...
EAL: VFIO support initialized
.....
EAL: PCI device 0000:07:00.1 on NUMA socket 0
EAL:   probe driver: 177d:a04b octeontx_ssovf
.....
EAL: PCI device 0001:02:00.7 on NUMA socket 0
EAL:   probe driver: 177d:a0dd octeontx_pkivf
.....
EAL: PCI device 0001:03:01.0 on NUMA socket 0
EAL:   probe driver: 177d:a049 octeontx_pkovf
.....
PMD: octeontx_probe(): created ethdev eth_octeontx for port 0
PMD: octeontx_probe(): created ethdev eth_octeontx for port 1
.....
Configuring Port 0 (socket 0)
Port 0: 00:0F:B7:11:94:46
Configuring Port 1 (socket 0)
Port 1: 00:0F:B7:11:94:47
.....
Checking link statuses...
Port 0 Link Up - speed 40000 Mbps - full-duplex
Port 1 Link Up - speed 40000 Mbps - full-duplex
Done
testpmd>
```

9.41.6 Initialization

The OCTEON TX ethdev pmd is exposed as a vdev device which consists of a set of PKI and PKO PCIe VF devices. On EAL initialization, PKI/PKO PCIe VF devices will be probed and then the vdev device can be created from the application code, or from the EAL command line based on the number of probed/bound PKI/PKO PCIe VF device to DPDK by

- Invoking `rte_vdev_init("eth_octeontx")` from the application
- Using `--vdev="eth_octeontx"` in the EAL options, which will call rte_vdev_init() internally

Device arguments

Each ethdev port is mapped to a physical port(LMAC), Application can specify the number of interesting ports with `nr_ports` argument.

Dependency

`eth_octeontx` pmd is depend on `event_octeontx` eventdev device and `octeontx_fpavf` external mempool handler.

Example:

```
./your_dpdk_application --mbuf-pool-ops-name="octeontx_fpavf" \
            --vdev='event_octeontx' \
            --vdev="eth_octeontx,nr_port=2"
```

9.41.7 Limitations

`octeontx_fpavf` external mempool handler dependency

The OCTEON TX SoC family NIC has inbuilt HW assisted external mempool manager. This driver will only work with `octeontx_fpavf` external mempool handler as it is the most performance effective way for packet allocation and Tx buffer recycling on OCTEON TX SoC platform.

CRC stripping

The OCTEON TX SoC family NICs strip the CRC for every packets coming into the host interface irrespective of the offload configuration.

Maximum packet length

The OCTEON TX SoC family NICs support a maximum of a 32K jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 32k, frames up to 32k bytes can still reach the host interface.

Maximum mempool size

The maximum mempool size supplied to Rx queue setup should be less than 128K. When running testpmd on OCTEON TX the application can limit the number of mbufs by using the option `--total-num-mbufs=131072`.

9.42 OCTEON TX2 Poll Mode driver

The OCTEON TX2 ETHDEV PMD (**librte_pmd_octeontx2**) provides poll mode ethdev driver support for the inbuilt network device found in **Marvell OCTEON TX2** SoC family as well as for their virtual functions (VF) in SR-IOV context.

More information can be found at Marvell Official Website.

9.42.1 Features

Features of the OCTEON TX2 Ethdev PMD are:

- Packet type information
- Promiscuous mode
- Jumbo frames
- SR-IOV VF
- Lock-free Tx queue
- Multiple queues for TX and RX
- Receiver Side Scaling (RSS)
- MAC/VLAN filtering
- Multicast MAC filtering
- Generic flow API
- Inner and Outer Checksum offload
- VLAN/QinQ stripping and insertion
- Port hardware statistics
- Link state information
- Link flow control
- MTU update
- Scatter-Gather IO support
- Vector Poll mode driver
- Debug utilities - Context dump and error interrupt support
- IEEE1588 timestamping
- HW offloaded *ethdev Rx queue* to *eventdev event queue* packet injection
- Support Rx interrupt

- Inline IPsec processing support
- *Traffic Management API*

9.42.2 Prerequisites

See *Marvell OCTEON TX2 Platform Guide* for setup information.

9.42.3 Compile time Config Options

The following options may be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_OCTEONTX2_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_octeontx2` driver.

9.42.4 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

To compile the OCTEON TX2 PMD for Linux arm64 gcc, use arm64-octeontx2-linux-gcc as target.

1. Running testpmd:

   Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

   Example output:

```
./build/app/testpmd -c 0x300 -w 0002:02:00.0 -- --portmask=0x1 --nb-cores=1 --port-
↪topology=loop --rxq=1 --txq=1
EAL: Detected 24 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: No available hugepages reported in hugepages-2048kB
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL: PCI device 0002:02:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a063 net_octeontx2
EAL:   using IOMMU type 1 (Type 1)
testpmd: create a new mbuf pool <mbuf_pool_socket_0>: n=267456, size=2176, socket=0
testpmd: preferred mempool ops selected: octeontx2_npa
Configuring Port 0 (socket 0)
PMD: Port 0: Link Up - speed 40000 Mbps - full-duplex

Port 0: link state change event
Port 0: 36:10:66:88:7A:57
Checking link statuses...
Done
No commandline core given, start packet forwarding
io packet forwarding - ports=1 - cores=1 - streams=1 - NUMA support enabled, MP␣
↪allocation mode: native
Logical Core 9 (socket 0) forwards packets on 1 streams:
  RX P=0/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer=02:00:00:00:00:00

  io packet forwarding packets/burst=32
  nb forwarding cores=1 - nb forwarding ports=1
  port 0: RX queue number: 1 Tx queue number: 1
```

(continues on next page)

(continued from previous page)

```
    Rx offloads=0x0 Tx offloads=0x10000
    RX queue: 0
      RX desc=512 - RX free threshold=0
      RX threshold registers: pthresh=0 hthresh=0  wthresh=0
      RX Offloads=0x0
    TX queue: 0
      TX desc=512 - TX free threshold=0
      TX threshold registers: pthresh=0 hthresh=0  wthresh=0
      TX offloads=0x10000 - TX RS bit threshold=0
Press enter to exit
```

9.42.5 Runtime Config Options

- `Rx&Tx scalar mode enable` (default `0`)

  Ethdev supports both scalar and vector mode, it may be selected at runtime using `scalar_enable` `devargs` parameter.

- `RSS reta size` (default `64`)

  RSS redirection table size may be configured during runtime using `reta_size` `devargs` parameter.

  For example:

  ```
  -w 0002:02:00.0,reta_size=256
  ```

  With the above configuration, reta table of size 256 is populated.

- `Flow priority levels` (default `3`)

  RTE Flow priority levels can be configured during runtime using `flow_max_priority` `devargs` parameter.

  For example:

  ```
  -w 0002:02:00.0,flow_max_priority=10
  ```

  With the above configuration, priority level was set to 10 (0-9). Max priority level supported is 32.

- `Reserve Flow entries` (default `8`)

  RTE flow entries can be pre allocated and the size of pre allocation can be selected runtime using `flow_prealloc_size` `devargs` parameter.

  For example:

  ```
  -w 0002:02:00.0,flow_prealloc_size=4
  ```

  With the above configuration, pre alloc size was set to 4. Max pre alloc size supported is 32.

- `Max SQB buffer count` (default `512`)

  Send queue descriptor buffer count may be limited during runtime using `max_sqb_count` `devargs` parameter.

  For example:

```
-w 0002:02:00.0,max_sqb_count=64
```

With the above configuration, each send queue's decscriptor buffer count is limited to a maximum of 64 buffers.

- `Switch header enable` (default `none`)

  A port can be configured to a specific switch header type by using `switch_header devargs` parameter.

  For example:

  ```
  -w 0002:02:00.0,switch_header="higig2"
  ```

  With the above configuration, higig2 will be enabled on that port and the traffic on this port should be higig2 traffic only. Supported switch header types are "higig2", "dsa" and "chlen90b".

- `RSS tag as XOR` (default `0`)

  C0 HW revision onward, The HW gives an option to configure the RSS adder as

  - `rss_adder<7:0> = flow_tag<7:0> ^ flow_tag<15:8> ^ flow_tag<23:16> ^ flow_tag<31:24>`
  - `rss_adder<7:0> = flow_tag<7:0>`

  Latter one aligns with standard NIC behavior vs former one is a legacy RSS adder scheme used in OCTEON TX2 products.

  By default, the driver runs in the latter mode from C0 HW revision onward. Setting this flag to 1 to select the legacy mode.

  For example to select the legacy mode(RSS tag adder as XOR):

  ```
  -w 0002:02:00.0,tag_as_xor=1
  ```

- `Max SPI for inbound inline IPsec` (default `1`)

  Max SPI supported for inbound inline IPsec processing can be specified by `ipsec_in_max_spi devargs` parameter.

  For example:

  ```
  -w 0002:02:00.0,ipsec_in_max_spi=128
  ```

  With the above configuration, application can enable inline IPsec processing on 128 SAs (SPI 0-127).

Note: Above devarg parameters are configurable per device, user needs to pass the parameters to all the PCIe devices if application requires to configure on all the ethdev ports.

- `Lock NPA contexts in NDC`

  Lock NPA aura and pool contexts in NDC cache. The device args take hexadecimal bitmask where each bit represent the corresponding aura/pool id.

  For example:

```
-w 0002:02:00.0,npa_lock_mask=0xf
```

9.42.6 Traffic Management API

OCTEON TX2 PMD supports generic DPDK Traffic Management API which allows to configure the following features:

1. Hierarchical scheduling
2. Single rate - Two color, Two rate - Three color shaping

Both DWRR and Static Priority(SP) hierarchial scheduling is supported.

Every parent can have atmost 10 SP Children and unlimited DWRR children.

Both PF & VF supports traffic management API with PF supporting 6 levels and VF supporting 5 levels of topology.

9.42.7 Limitations

`mempool_octeontx2` external mempool handler dependency

The OCTEON TX2 SoC family NIC has inbuilt HW assisted external mempool manager. `net_octeontx2` pmd only works with `mempool_octeontx2` mempool handler as it is performance wise most effective way for packet allocation and Tx buffer recycling on OCTEON TX2 SoC platform.

CRC stripping

The OCTEON TX2 SoC family NICs strip the CRC for every packet being received by the host interface irrespective of the offload configuration.

Multicast MAC filtering

`net_octeontx2` pmd supports multicast mac filtering feature only on physical function devices.

SDP interface support

OCTEON TX2 SDP interface support is limited to PF device, No VF support.

Inline Protocol Processing

`net_octeontx2` pmd doesn't support the following features for packets to be inline protocol processed. - TSO offload - VLAN/QinQ offload - Fragmentation

9.42.8 Debugging Options

Table 9.12: OCTEON TX2 ethdev debug options

#	Component	EAL log command
1	NIX	–log-level='pmd.net.octeontx2,8'
2	NPC	–log-level='pmd.net.octeontx2.flow,8'

9.42.9 RTE Flow Support

The OCTEON TX2 SoC family NIC has support for the following patterns and actions.

Patterns:

Table 9.13: Item types

#	Pattern Type
1	RTE_FLOW_ITEM_TYPE_ETH
2	RTE_FLOW_ITEM_TYPE_VLAN
3	RTE_FLOW_ITEM_TYPE_E_TAG
4	RTE_FLOW_ITEM_TYPE_IPV4
5	RTE_FLOW_ITEM_TYPE_IPV6
6	RTE_FLOW_ITEM_TYPE_ARP_ETH_IPV4
7	RTE_FLOW_ITEM_TYPE_MPLS
8	RTE_FLOW_ITEM_TYPE_ICMP
9	RTE_FLOW_ITEM_TYPE_UDP
10	RTE_FLOW_ITEM_TYPE_TCP
11	RTE_FLOW_ITEM_TYPE_SCTP
12	RTE_FLOW_ITEM_TYPE_ESP
13	RTE_FLOW_ITEM_TYPE_GRE
14	RTE_FLOW_ITEM_TYPE_NVGRE
15	RTE_FLOW_ITEM_TYPE_VXLAN
16	RTE_FLOW_ITEM_TYPE_GTPC
17	RTE_FLOW_ITEM_TYPE_GTPU
18	RTE_FLOW_ITEM_TYPE_GENEVE
19	RTE_FLOW_ITEM_TYPE_VXLAN_GPE
20	RTE_FLOW_ITEM_TYPE_IPV6_EXT
21	RTE_FLOW_ITEM_TYPE_VOID
22	RTE_FLOW_ITEM_TYPE_ANY
23	RTE_FLOW_ITEM_TYPE_GRE_KEY
24	RTE_FLOW_ITEM_TYPE_HIGIG2

Note: `RTE_FLOW_ITEM_TYPE_GRE_KEY` works only when checksum and routing bits in the GRE header are equal to 0.

Actions:

Table 9.14: Ingress action types

#	Action Type
1	RTE_FLOW_ACTION_TYPE_VOID
2	RTE_FLOW_ACTION_TYPE_MARK
3	RTE_FLOW_ACTION_TYPE_FLAG
4	RTE_FLOW_ACTION_TYPE_COUNT
5	RTE_FLOW_ACTION_TYPE_DROP
6	RTE_FLOW_ACTION_TYPE_QUEUE
7	RTE_FLOW_ACTION_TYPE_RSS
8	RTE_FLOW_ACTION_TYPE_SECURITY
9	RTE_FLOW_ACTION_TYPE_PF
10	RTE_FLOW_ACTION_TYPE_VF

Table 9.15: Egress action types

#	Action Type
1	RTE_FLOW_ACTION_TYPE_COUNT
2	RTE_FLOW_ACTION_TYPE_DROP

9.43 PFE Poll Mode Driver

The PFE NIC PMD (**librte_pmd_pfe**) provides poll mode driver support for the inbuilt NIC found in the **NXP LS1012** SoC.

More information can be found at NXP Official Website.

9.43.1 PFE

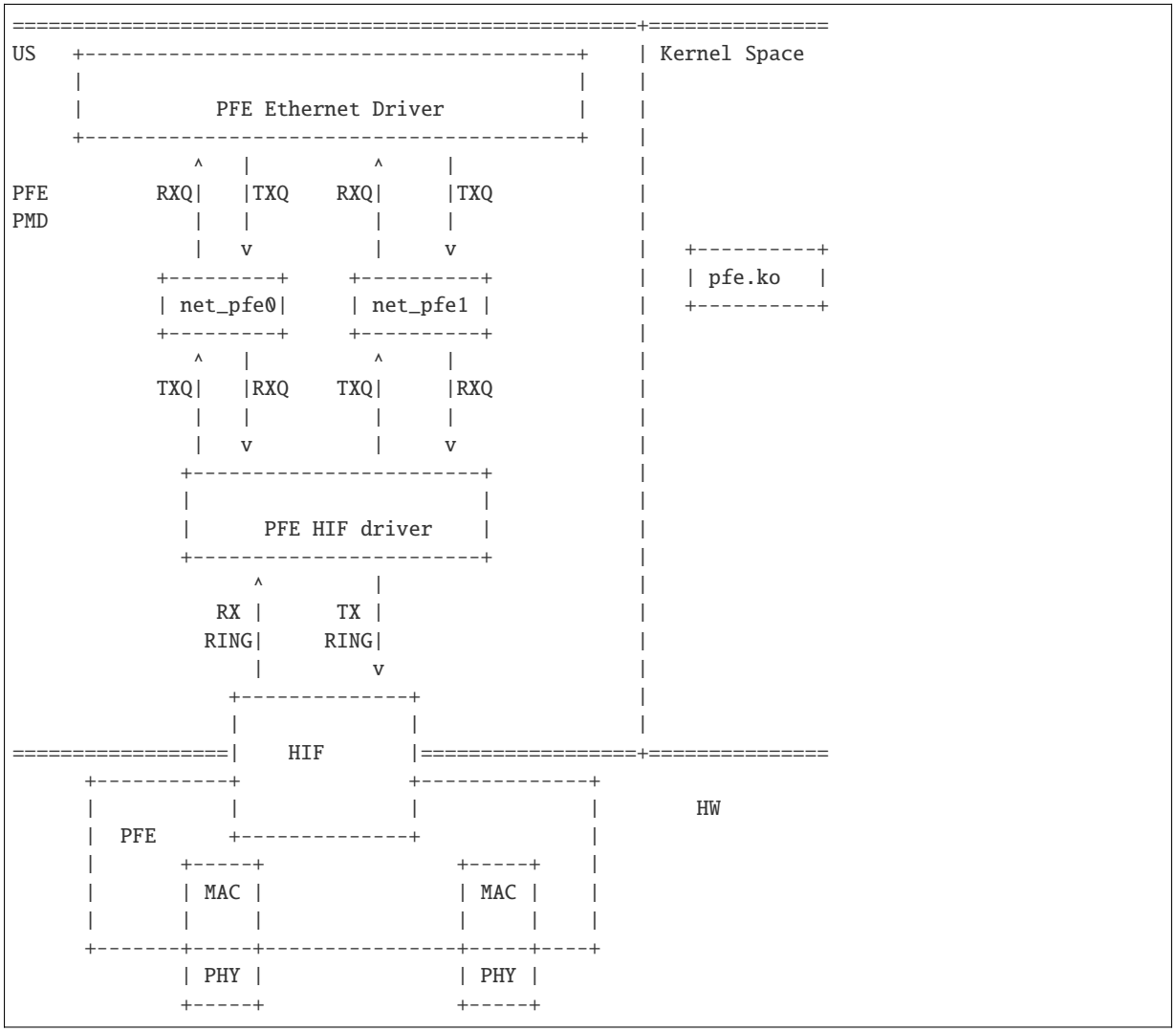
This section provides an overview of the NXP PFE and how it is integrated into the DPDK.

Contents summary

- PFE overview
- PFE features
- Supported PFE SoCs
- Prerequisites
- Driver compilation and testing
- Limitations

PFE Overview

PFE is a hardware programmable packet forwarding engine to provide high performance Ethernet interfaces. The diagram below shows a system level overview of PFE:



The HIF, PFE, MAC and PHY are the hardware blocks, the pfe.ko is a kernel module, the PFE HIF driver and the PFE ethernet driver combined represent as DPDK PFE poll mode driver are running in the userspace.

The PFE hardware supports one HIF (host interface) RX ring and one TX ring to send and receive packets through packet forwarding engine. Both network interface traffic is multiplexed and send over HIF queue.

net_pfe0 and net_pfe1 are logical ethernet interfaces, created by HIF client driver. HIF driver is responsible for send and receive packets between host interface and these logical interfaces. PFE ethernet driver is a hardware independent and register with the HIF client driver to transmit and receive packets from HIF via logical interfaces.

pfe.ko is required for PHY initialisation and also responsible for creating the character device "pfe_us_cdev" which will be used for interacting with the kernel layer for link status.

PFE Features

- L3/L4 checksum offload
- Packet type parsing
- Basic stats
- MTU update
- Promiscuous mode
- Allmulticast mode
- Link status
- ARMv8

Supported PFE SoCs

- LS1012

Prerequisites

Below are some pre-requisites for executing PFE PMD on a PFE compatible board:

1. **ARM 64 Tool Chain**

   For example, the *aarch64* Linaro Toolchain.

2. **Linux Kernel**

   It can be obtained from NXP's Github hosting.

3. **Rootfile system**

   Any *aarch64* supporting filesystem can be used. For example, Ubuntu 16.04 LTS (Xenial) or 18.04 (Bionic) userland which can be obtained from here.

4. The ethernet device will be registered as virtual device, so pfe has dependency on **rte_bus_vdev** library and it is mandatory to use *–vdev* with value *net_pfe* to run DPDK application.

The following dependencies are not part of DPDK and must be installed separately:

- **NXP Linux LSDK**

  NXP Layerscape software development kit (LSDK) includes support for family of QorIQ® ARM-Architecture-based system on chip (SoC) processors and corresponding boards.

  It includes the Linux board support packages (BSPs) for NXP SoCs, a fully operational tool chain, kernel and board specific modules.

  LSDK and related information can be obtained from: LSDK

- **pfe kernel module**

  pfe kernel module can be obtained from NXP Layerscape software development kit at location */lib/modules/<kernel version>/kernel/drivers/staging/fsl_ppfe* in rootfs. Module should be loaded using below command:

```
insmod pfe.ko us=1
```

Driver compilation and testing

Follow instructions available in the document *compiling and testing a PMD for a NIC* to launch **testpmd**

Additionally, PFE driver needs *–vdev* as an input with value *net_pfe* to execute DPDK application. There is an optional parameter *intf* available to specify port ID. PFE driver supports only two interfaces, so valid values for *intf* are 0 and 1. see the command below:

```
<dpdk app> <EAL args> --vdev="net_pfe0,intf=0" --vdev="net_pfe1,intf=1" -- ...
```

Limitations

- Multi buffer pool cannot be supported.

9.44 QEDE Poll Mode Driver

The QEDE poll mode driver library (**librte_pmd_qede**) implements support for **QLogic FastLinQ QL4xxxx 10G/25G/40G/50G/100G Intelligent Ethernet Adapters (IEA) and Converged Network Adapters (CNA)** family of adapters as well as SR-IOV virtual functions (VF). It is supported on several standard Linux distros like RHEL, SLES, Ubuntu etc. It is compile-tested under FreeBSD OS.

More information can be found at QLogic Corporation's Website.

9.44.1 Supported Features

- Unicast/Multicast filtering
- Promiscuous mode
- Allmulti mode
- Port hardware statistics
- Jumbo frames
- Multiple MAC address
- MTU change
- Default pause flow control
- Multiprocess aware
- Scatter-Gather
- Multiple Rx/Tx queues
- RSS (with RETA/hash table/key)
- TSS
- Stateless checksum offloads (IPv4/IPv6/TCP/UDP)

- LRO/TSO
- VLAN offload - Filtering and stripping
- N-tuple filter and flow director (limited support)
- NPAR (NIC Partitioning)
- SR-IOV VF
- GRE Tunneling offload
- GENEVE Tunneling offload
- VXLAN Tunneling offload
- MPLSoUDP Tx Tunneling offload
- Generic flow API

9.44.2 Non-supported Features

- SR-IOV PF

9.44.3 Co-existence considerations

- QLogic FastLinQ QL4xxxx CNAs support Ethernet, RDMA, iSCSI and FCoE functionalities. These functionalities are supported using QLogic Linux kernel drivers qed, qede, qedr, qedi and qedf. DPDK is supported on these adapters using qede PMD.
- When SR-IOV is not enabled on the adapter, QLogic Linux kernel drivers (qed, qede, qedr, qedi and qedf) and qede PMD can't be attached to different PFs on a given QLogic FastLinQ QL4xxx adapter. A given adapter needs to be completely used by DPDK or Linux drivers Before binding DPDK driver to one or more PFs on the adapter, please make sure to unbind Linux drivers from all PFs of the adapter. If there are multiple adapters on the system, one or more adapters can be used by DPDK driver completely and other adapters can be used by Linux drivers completely.
- When SR-IOV is enabled on the adapter, Linux kernel drivers (qed, qede, qedr, qedi and qedf) can be bound to the PFs of a given adapter and either qede PMD or Linux drivers (qed and qede) can be bound to the VFs of the adapter.
- For sharing an adapter between DPDK and Linux drivers, SRIOV needs to be enabled. Bind all the PFs to Linux Drivers(qed/qede). Create a VF on PFs where DPDK is desired and bind these VFs to qede_pmd. Binding of PFs simultaneously to DPDK and Linux drivers on a given adapter is not supported.

9.44.4 Supported QLogic Adapters

- QLogic FastLinQ QL4xxxx 10G/25G/40G/50G/100G Intelligent Ethernet Adapters (IEA) and Converged Network Adapters (CNA)

9.44.5 Prerequisites

- Requires storm firmware version **8.40.33.0**. Firmware may be available inbox in certain newer Linux distros under the standard directory `E.g. /lib/firmware/qed/qed_init_values-8.40.33.0.bin`. If the required firmware files are not available then download it from linux-firmware git repository.
- Requires the NIC be updated minimally with **8.30.x.x** Management firmware(MFW) version supported for that NIC. It is highly recommended that the NIC be updated with the latest available management firmware version to get latest feature set. Management Firmware and Firmware Upgrade Utility for Cavium FastLinQ(r) branded adapters can be downloaded from Driver Download Center. For downloading Firmware Upgrade Utility, select NIC category, model and Linux distro. To update the management firmware, refer to the instructions in the Firmware Upgrade Utility Readme document. For OEM branded adapters please follow the instruction provided by the OEM to update the Management Firmware on the NIC.
- SR-IOV requires Linux PF driver version **8.20.x.x** or higher. If the required PF driver is not available then download it from QLogic Driver Download Center. For downloading PF driver, select adapter category, model and Linux distro.

Performance note

- For better performance, it is recommended to use 4K or higher RX/TX rings.

Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_QEDE_PMD` (default **y**)

  Toggle compilation of QEDE PMD driver.

- `CONFIG_RTE_LIBRTE_QEDE_DEBUG_TX` (default **n**)

  Toggle display of transmit fast path run-time messages.

- `CONFIG_RTE_LIBRTE_QEDE_DEBUG_RX` (default **n**)

  Toggle display of receive fast path run-time messages.

- `CONFIG_RTE_LIBRTE_QEDE_FW` (default **“”**)

  Gives absolute path of firmware file. `Eg: "/lib/firmware/qed/qed_init_values-8.40.33.0.bin"` Empty string indicates driver will pick up the firmware file from the default location /lib/firmware/qed. CAUTION this option is more for custom firmware, it is not recommended for use under normal condition.

Config notes

When there are multiple adapters and/or large number of Rx/Tx queues configured on the adapters, the default (2560) number of memzone descriptors may not be enough. Please increase the number of memzone descriptors to a higher number as needed. When sufficient number of memzone descriptors are not configured, user can potentially run into following error.

```
EAL: memzone_reserve_aligned_thread_unsafe(): No more room in config
```

9.44.6 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

9.44.7 RTE Flow Support

QLogic FastLinQ QL4xxxx NICs has support for the following patterns and actions.

Patterns:

Table 9.16: Item types

#	Pattern Type
1	RTE_FLOW_ITEM_TYPE_IPV4
2	RTE_FLOW_ITEM_TYPE_IPV6
3	RTE_FLOW_ITEM_TYPE_UDP
4	RTE_FLOW_ITEM_TYPE_TCP

Actions:

Table 9.17: Ingress action types

#	Action Type
1	RTE_FLOW_ACTION_TYPE_QUEUE
2	RTE_FLOW_ACTION_TYPE_DROP

9.44.8 SR-IOV: Prerequisites and Sample Application Notes

This section provides instructions to configure SR-IOV with Linux OS.

Note: librte_pmd_qede will be used to bind to SR-IOV VF device and Linux native kernel driver (qede) will function as SR-IOV PF driver. Requires PF driver to be 8.20.x.x or higher.

1. Verify SR-IOV and ARI capability is enabled on the adapter using `lspci`:

```
lspci -s <slot> -vvv
```

Example output:

```
[...]
Capabilities: [1b8 v1] Alternative Routing-ID Interpretation (ARI)
[...]
Capabilities: [1c0 v1] Single Root I/O Virtualization (SR-IOV)
```

(continues on next page)

(continued from previous page)

```
[...]
Kernel driver in use: igb_uio
```

2. Load the kernel module:

```
modprobe qede
```

Example output:

```
systemd-udevd[4848]: renamed network interface eth0 to ens5f0
systemd-udevd[4848]: renamed network interface eth1 to ens5f1
```

3. Bring up the PF ports:

```
ifconfig ens5f0 up
ifconfig ens5f1 up
```

4. Create VF device(s):

Echo the number of VFs to be created into "sriov_numvfs" sysfs entry of the parent PF.

Example output:

```
echo 2 > /sys/devices/pci0000:00/0000:00:03.0/0000:81:00.0/sriov_numvfs
```

5. Assign VF MAC address:

Assign MAC address to the VF using iproute2 utility. The syntax is:

```
ip link set <PF iface> vf <VF id> mac <macaddr>
```

Example output:

```
ip link set ens5f0 vf 0 mac 52:54:00:2f:9d:e8
```

6. PCI Passthrough:

The VF devices may be passed through to the guest VM using virt-manager or virsh. QEDE PMD should be used to bind the VF devices in the guest VM using the instructions from Driver compilation and testing section above.

7. Running testpmd (Supply --log-level="pmd.net.qede.driver:info to view informational messages):

Refer to the document *compiling and testing a PMD for a NIC* to run testpmd application.

Example output:

```
  testpmd -l 0,4-11 -n 4 -- -i --nb-cores=8 --portmask=0xf --rxd=4096 \
  --txd=4096 --txfreet=4068 --enable-rx-cksum --rxq=4 --txq=4 \
  --rss-ip --rss-udp

  [...]

EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL:   probe driver: 1077:1634 rte_qede_pmd
EAL:   Not managed by a supported kernel driver, skipped
EAL: PCI device 0000:84:00.1 on NUMA socket 1
```

(continues on next page)

(continued from previous page)

```
EAL:   probe driver: 1077:1634 rte_qede_pmd
EAL:   Not managed by a supported kernel driver, skipped
EAL: PCI device 0000:88:00.0 on NUMA socket 1
EAL:   probe driver: 1077:1656 rte_qede_pmd
EAL:   PCI memory mapped at 0x7f738b200000
EAL:   PCI memory mapped at 0x7f738b280000
EAL:   PCI memory mapped at 0x7f738b300000
PMD: Chip details : BB1
PMD: Driver version : QEDE PMD 8.7.9.0_1.0.0
PMD: Firmware version : 8.7.7.0
PMD: Management firmware version : 8.7.8.0
PMD: Firmware file : /lib/firmware/qed/qed_init_values_zipped-8.7.7.0.bin
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_common_dev_init:macaddr \
                                                    00:0e:1e:d2:09:9c
  [...]
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_tx_queue_setup:txq 0 num_desc 4096 \
                                           tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_tx_queue_setup:txq 1 num_desc 4096 \
                                           tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_tx_queue_setup:txq 2 num_desc 4096 \
                                            tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_tx_queue_setup:txq 3 num_desc 4096 \
                                            tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_rx_queue_setup:rxq 0 num_desc 4096 \
                                           rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_rx_queue_setup:rxq 1 num_desc 4096 \
                                           rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_rx_queue_setup:rxq 2 num_desc 4096 \
                                           rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_rx_queue_setup:rxq 3 num_desc 4096 \
                                           rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_dev_start:port 0
[QEDE PMD: (84:00.0:dpdk-port-0)]qede_dev_start:link status: down
  [...]
Checking link statuses...
Port 0 Link Up - speed 25000 Mbps - full-duplex
Port 1 Link Up - speed 25000 Mbps - full-duplex
Port 2 Link Up - speed 25000 Mbps - full-duplex
Port 3 Link Up - speed 25000 Mbps - full-duplex
Done
testpmd>
```

9.45 Solarflare libefx-based Poll Mode Driver

The SFC EFX PMD (**librte_pmd_sfc_efx**) provides poll mode driver support for **Solarflare SFN7xxx and SFN8xxx** family of 10/40 Gbps adapters and **Solarflare XtremeScale X2xxx** family of 10/25/40/50/100 Gbps adapters. SFC EFX PMD has support for the latest Linux and FreeBSD operating systems.

More information can be found at Solarflare Communications website.

9.45.1 Features

SFC EFX PMD has support for:

- Multiple transmit and receive queues
- Link state information including link status change interrupt
- IPv4/IPv6 TCP/UDP transmit checksum offload
- Inner IPv4/IPv6 TCP/UDP transmit checksum offload
- Port hardware statistics
- Extended statistics (see Solarflare Server Adapter User's Guide for the statistics description)
- Basic flow control
- MTU update
- Jumbo frames up to 9K
- Promiscuous mode
- Allmulticast mode
- TCP segmentation offload (TSO) including VXLAN and GENEVE encapsulated
- Multicast MAC filter
- IPv4/IPv6 TCP/UDP receive checksum offload
- Inner IPv4/IPv6 TCP/UDP receive checksum offload
- Received packet type information
- Receive side scaling (RSS)
- RSS hash
- Scattered Rx DMA for packet that are larger that a single Rx descriptor
- Receive queue interrupts
- Deferred receive and transmit queue start
- Transmit VLAN insertion (if running firmware variant supports it)
- Flow API
- Loopback

9.45.2 Non-supported Features

The features not yet supported include:

- Priority-based flow control
- Configurable RX CRC stripping (always stripped)
- Header split on receive
- VLAN filtering
- VLAN stripping

- LRO

9.45.3 Limitations

Due to requirements on receive buffer alignment and usage of the receive buffer for the auxiliary packet information provided by the NIC up to extra 269 (14 bytes prefix plus up to 255 bytes for end padding) bytes may be required in the receive buffer. It should be taken into account when mbuf pool for receive is created.

Equal stride super-buffer mode

When the receive queue uses equal stride super-buffer DMA mode, one HW Rx descriptor carries many Rx buffers which contiguously follow each other with some stride (equal to total size of rte_mbuf as mempool object). Each Rx buffer is an independent rte_mbuf. However dedicated mempool manager must be used when mempool for the Rx queue is created. The manager must support dequeue of the contiguous block of objects and provide mempool info API to get the block size.

Another limitation of a equal stride super-buffer mode, imposed by the firmware, is that it allows for a single RSS context.

9.45.4 Tunnels support

NVGRE, VXLAN and GENEVE tunnels are supported on SFN8xxx and X2xxx family adapters with full-feature firmware variant running. **sfboot** should be used to configure NIC to run full-feature firmware variant. See Solarflare Server Adapter User's Guide for details.

SFN8xxx and X2xxx family adapters provide either inner or outer packet classes. If adapter firmware advertises support for tunnels then the PMD configures the hardware to report inner classes, and outer classes are not reported in received packets. However, for VXLAN and GENEVE tunnels the PMD does report UDP as the outer layer 4 packet type.

SFN8xxx and X2xxx family adapters report GENEVE packets as VXLAN. If UDP ports are configured for only one tunnel type then it is safe to treat VXLAN packet type indication as the corresponding UDP tunnel type.

9.45.5 Flow API support

Supported attributes:

- Ingress

Supported pattern items:

- VOID
- ETH (exact match of source/destination addresses, individual/group match of destination address, EtherType in the outer frame and exact match of destination addresses, individual/group match of destination address in the inner frame)
- VLAN (exact match of VID, double-tagging is supported)
- IPV4 (exact match of source/destination addresses, IP transport protocol)
- IPV6 (exact match of source/destination addresses, IP transport protocol)

- TCP (exact match of source/destination ports)
- UDP (exact match of source/destination ports)
- VXLAN (exact match of VXLAN network identifier)
- GENEVE (exact match of virtual network identifier, only Ethernet (0x6558) protocol type is supported)
- NVGRE (exact match of virtual subnet ID)

Supported actions:

- VOID
- QUEUE
- RSS
- DROP
- FLAG (supported only with ef10_essb Rx datapath)
- MARK (supported only with ef10_essb Rx datapath)

Validating flow rules depends on the firmware variant.

The *Flow isolated mode* is supported.

Ethernet destination individual/group match

Ethernet item supports I/G matching, if only the corresponding bit is set in the mask of destination address. If destination address in the spec is multicast, it matches all multicast (and broadcast) packets, otherwise it matches unicast packets that are not filtered by other flow rules.

Exceptions to flow rules

There is a list of exceptional flow rule patterns which will not be accepted by the PMD. A pattern will be rejected if at least one of the conditions is met:

- Filtering by IPv4 or IPv6 EtherType without pattern items of internet layer and above.
- The last item is IPV4 or IPV6, and it's empty.
- Filtering by TCP or UDP IP transport protocol without pattern items of transport layer and above.
- The last item is TCP or UDP, and it's empty.

9.45.6 Supported NICs

- Solarflare XtremeScale Adapters:
  - Solarflare X2522 Dual Port SFP28 10/25GbE Adapter
  - Solarflare X2541 Single Port QSFP28 10/25G/100G Adapter
  - Solarflare X2542 Dual Port QSFP28 10/25G/100G Adapter
- Solarflare Flareon [Ultra] Server Adapters:

- Solarflare SFN8522 Dual Port SFP+ Server Adapter
- Solarflare SFN8522M Dual Port SFP+ Server Adapter
- Solarflare SFN8042 Dual Port QSFP+ Server Adapter
- Solarflare SFN8542 Dual Port QSFP+ Server Adapter
- Solarflare SFN8722 Dual Port SFP+ OCP Server Adapter
- Solarflare SFN7002F Dual Port SFP+ Server Adapter
- Solarflare SFN7004F Quad Port SFP+ Server Adapter
- Solarflare SFN7042Q Dual Port QSFP+ Server Adapter
- Solarflare SFN7122F Dual Port SFP+ Server Adapter
- Solarflare SFN7124F Quad Port SFP+ Server Adapter
- Solarflare SFN7142Q Dual Port QSFP+ Server Adapter
- Solarflare SFN7322F Precision Time Synchronization Server Adapter

9.45.7 Prerequisites

- Requires firmware version:
  - SFN7xxx: **4.7.1.1001** or higher
  - SFN8xxx: **6.0.2.1004** or higher

Visit Solarflare Support Downloads to get Solarflare Utilities (either Linux or FreeBSD) with the latest firmware. Follow instructions from Solarflare Server Adapter User's Guide to update firmware and configure the adapter.

9.45.8 Pre-Installation Configuration

Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_SFC_EFX_PMD` (default **y**)

  Enable compilation of Solarflare libefx-based poll-mode driver.

- `CONFIG_RTE_LIBRTE_SFC_EFX_DEBUG` (default **n**)

  Enable compilation of the extra run-time consistency checks.

Per-Device Parameters

The following per-device parameters can be passed via EAL PCI device whitelist option like "-w 02:00.0,arg1=value1,...".

Case-insensitive 1/y/yes/on or 0/n/no/off may be used to specify boolean parameters value.

- `rx_datapath` [auto|efx|ef10|ef10_esps] (default **auto**)

  Choose receive datapath implementation. **auto** allows the driver itself to make a choice based on firmware features available and required by the datapath implementation. **efx** chooses libefx-based datapath which supports Rx scatter. **ef10** chooses EF10 (SFN7xxx, SFN8xxx, X2xxx) native datapath which is more efficient than libefx-based and provides richer packet type classification. **ef10_esps** chooses SFNX2xxx equal stride packed stream datapath which may be used on DPDK firmware variant only (see notes about its limitations above).

- `tx_datapath` [auto|efx|ef10|ef10_simple] (default **auto**)

  Choose transmit datapath implementation. **auto** allows the driver itself to make a choice based on firmware features available and required by the datapath implementation. **efx** chooses libefx-based datapath which supports VLAN insertion (full-feature firmware variant only), TSO and multi-segment mbufs. Mbuf segments may come from different mempools, and mbuf reference counters are treated responsibly. **ef10** chooses EF10 (SFN7xxx, SFN8xxx, X2xxx) native datapath which is more efficient than libefx-based but has no VLAN insertion support yet. Mbuf segments may come from different mempools, and mbuf reference counters are treated responsibly. **ef10_simple** chooses EF10 (SFN7xxx, SFN8xxx, X2xxx) native datapath which is even more faster then **ef10** but does not support multi-segment mbufs, disallows multiple mempools and neglects mbuf reference counters.

- `perf_profile` [auto|throughput|low-latency] (default **throughput**)

  Choose hardware tuning to be optimized for either throughput or low-latency. **auto** allows NIC firmware to make a choice based on installed licenses and firmware variant configured using **sfboot**.

- `stats_update_period_ms` [long] (default **1000**)

  Adjust period in milliseconds to update port hardware statistics. The accepted range is 0 to 65535. The value of **0** may be used to disable periodic statistics update. One should note that it's only possible to set an arbitrary value on SFN8xxx and X2xxx provided that firmware version is 6.2.1.1033 or higher, otherwise any positive value will select a fixed update period of **1000** milliseconds

- `fw_variant` [dont-care|full-feature|ultra-low-latency| capture-packed-stream|dpdk] (default **dont-care**)

  Choose the preferred firmware variant to use. In order for the selected option to have an effect, the **sfboot** utility must be configured with the **auto** firmware-variant option. The preferred firmware variant applies to all ports on the NIC. **dont-care** ensures that the driver can attach to an unprivileged function. The datapath firmware type to use is controlled by the **sfboot** utility. **full-feature** chooses full featured firmware. **ultra-low-latency** chooses firmware with fewer features but lower latency. **capture-packed-stream** chooses firmware for SolarCapture packed stream mode. **dpdk** chooses DPDK firmware with equal stride super-buffer Rx mode for higher Rx packet rate and packet marks support and firmware subvariant without checksumming on transmit for higher Tx packet rate if checksumming is not required.

- `rxd_wait_timeout_ns` [long] (default **200 us**)

  Adjust timeout in nanoseconds to head-of-line block to wait for Rx descriptors. The accepted range is 0 to 400 ms. Flow control should be enabled to make it work. The value of **0** disables

it and packets are dropped immediately. When a packet is dropped because of no Rx descriptors, `rx_nodesc_drop_cnt` counter grows. The feature is supported only by the DPDK firmware variant when equal stride super-buffer Rx mode is used.

Dynamic Logging Parameters

One may leverage EAL option "–log-level" to change default levels for the log types supported by the driver. The option is used with an argument typically consisting of two parts separated by a colon.

Level value is the last part which takes a symbolic name (or integer). Log type is the former part which may shell match syntax. Depending on the choice of the expression, the given log level may be used either for some specific log type or for a subset of types.

SFC EFX PMD provides the following log types available for control:

- `pmd.net.sfc.driver` (default level is **notice**)

  Affects driver-wide messages unrelated to any particular devices.

- `pmd.net.sfc.main` (default level is **notice**)

  Matches a subset of per-port log types registered during runtime. A full name for a particular type may be obtained by appending a dot and a PCI device identifier (`XXXX:XX:XX.X`) to the prefix.

- `pmd.net.sfc.mcdi` (default level is **notice**)

  Extra logging of the communication with the NIC's management CPU. The format of the log is consumed by the Solarflare netlogdecode cross-platform tool. May be managed per-port, as explained above.

9.46 Soft NIC Poll Mode Driver

The Soft NIC allows building custom NIC pipelines in software. The Soft NIC pipeline is DIY and reconfigurable through `firmware` (DPDK Packet Framework script).

The Soft NIC leverages the DPDK Packet Framework libraries (librte_port, librte_table and librte_pipeline) to make it modular, flexible and extensible with new functionality. Please refer to DPDK Programmer's Guide, Chapter `Packet Framework` and DPDK Sample Application User Guide, Chapter `IP Pipeline Application` for more details.

The Soft NIC is configured through the standard DPDK ethdev API (ethdev, flow, QoS, security). The internal framework is not externally visible.

Key benefits:

- Can be used to augment missing features to HW NICs.
- Allows consumption of advanced DPDK features without application redesign.
- Allows out-of-the-box performance boost of DPDK consumers applications simply by instantiating this type of Ethernet device.

9.46.1 Flow

- `Device creation`: Each Soft NIC instance is a virtual device.
- `Device start`: The Soft NIC firmware script is executed every time the device is started. The firmware script typically creates several internal objects, such as: memory pools, SW queues, traffic manager, action profiles, pipelines, etc.
- `Device stop`: All the internal objects that were previously created by the firmware script during device start are now destroyed.
- `Device run`: Each Soft NIC device needs one or several CPU cores to run. The firmware script maps each internal pipeline to a CPU core. Multiple pipelines can be mapped to the same CPU core. In order for a given pipeline assigned to CPU core X to run, the application needs to periodically call on CPU core X the *rte_pmd_softnic_run()* function for the current Soft NIC device.
- `Application run`: The application reads packets from the Soft NIC device RX queues and writes packets to the Soft NIC device TX queues.

9.46.2 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in `System Requirements` section of *the DPDK documentation* or refer to *DPDK Release Notes*.

9.46.3 Build options

The default PMD configuration available in the common_linux configuration file:

CONFIG_RTE_LIBRTE_PMD_SOFTNIC=y

Once the DPDK is built, all the DPDK applications include support for the Soft NIC PMD.

9.46.4 Soft NIC PMD arguments

The user can specify below arguments in EAL `--vdev` options to create the Soft NIC device instance:

–vdev “net_softnic0,firmware=firmware.cli,conn_port=8086”

1. `firmware`: path to the firmware script used for Soft NIC configuration. The example “firmware” script is provided at *drivers/net/softnic/*. (Optional: No, Default = NA)
2. `conn_port`: tcp connection port (non-zero value) used by remote client (for examples- telnet, netcat, etc.) to connect and configure Soft NIC device in run-time. (Optional: yes, Default value: 0, no connection with external client)
3. `cpu_id`: numa node id. (Optional: yes, Default value: 0)
4. `tm_n_queues`: number of traffic manager’s scheduler queues. The traffic manager is based on DPDK *librte_sched* library. (Optional: yes, Default value: 65,536 queues)
5. `tm_qsize0`: size of scheduler queue 0 per traffic class of the pipes/subscribers. (Optional: yes, Default: 64)
6. `tm_qsize1`: size of scheduler queue 1 per traffic class of the pipes/subscribers. (Optional: yes, Default: 64)

7. `tm_qsize2`: size of scheduler queue 2 per traffic class of the pipes/subscribers. (Optional: yes, Default: 64)

8. `tm_qsize3`: size of scheduler queue 3 per traffic class of the pipes/subscribers. (Optional: yes, Default: 64)

9.46.5 Soft NIC testing

- Run testpmd application in Soft NIC forwarding mode with loopback feature enabled on Soft NIC port:

```
./testpmd -c 0x3 --vdev 'net_softnic0,firmware=<script path>/firmware.cli,cpu_
↪id=0,conn_port=8086' -- -i
     --forward-mode=softnic --portmask=0x2
```

```
...
Interactive-mode selected
Set softnic packet forwarding mode
...
Configuring Port 0 (socket 0)
Port 0: 90:E2:BA:37:9D:DC
Configuring Port 1 (socket 0)

; SPDX-License-Identifier: BSD-3-Clause
; Copyright(c) 2018 Intel Corporation

link LINK dev 0000:02:00.0

pipeline RX period 10 offset_port_id 0
pipeline RX port in bsz 32 link LINK rxq 0
pipeline RX port out bsz 32 swq RXQ0
pipeline RX table match stub
pipeline RX port in 0 table 0

pipeline TX period 10 offset_port_id 0
pipeline TX port in bsz 32 swq TXQ0
pipeline TX port out bsz 32 link LINK txq 0
pipeline TX table match stub
pipeline TX port in 0 table 0

thread 1 pipeline RX enable
thread 1 pipeline TX enable
Port 1: 00:00:00:00:00:00
Checking link statuses...
Done
testpmd>
```

- Start forwarding

```
testpmd> start
softnic packet forwarding - ports=1 - cores=1 - streams=1 - NUMA support␣
↪enabled, MP over anonymous pages disabled
Logical Core 1 (socket 0) forwards packets on 1 streams:
RX P=2/Q=0 (socket 0) -> TX P=2/Q=0 (socket 0) peer=02:00:00:00:00:02

softnic packet forwarding packets/burst=32
nb forwarding cores=1 - nb forwarding ports=1
port 0: RX queue number: 1 Tx queue number: 1
Rx offloads=0x1000 Tx offloads=0x0
```

(continues on next page)

(continued from previous page)

```
RX queue: 0
RX desc=512 - RX free threshold=32
RX threshold registers: pthresh=8 hthresh=8  wthresh=0
RX Offloads=0x0
TX queue: 0
TX desc=512 - TX free threshold=32
TX threshold registers: pthresh=32 hthresh=0  wthresh=0
TX offloads=0x0 - TX RS bit threshold=32
port 1: RX queue number: 1 Tx queue number: 1
Rx offloads=0x0 Tx offloads=0x0
RX queue: 0
RX desc=0 - RX free threshold=0
RX threshold registers: pthresh=0 hthresh=0  wthresh=0
RX Offloads=0x0
TX queue: 0
TX desc=0 - TX free threshold=0
TX threshold registers: pthresh=0 hthresh=0  wthresh=0
TX offloads=0x0 - TX RS bit threshold=0
```

- Start remote client (e.g. telnet) to communicate with the softnic device:

```
$ telnet 127.0.0.1 8086
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

Welcome to Soft NIC!

softnic>
```

- Add/update Soft NIC pipeline table match-action entries from telnet client:

```
softnic> pipeline RX table 0 rule add match default action fwd port 0
softnic> pipeline TX table 0 rule add match default action fwd port 0
```

9.46.6 Soft NIC Firmware

The Soft NIC firmware, for example- *softnic/firmware.cli*, consists of following CLI commands for creating and managing software based NIC pipelines. For more details, please refer to CLI command description provided in *softnic/rte_eth_softnic_cli.c*.

- Physical port for packets send/receive:

```
link LINK dev 0000:02:00.0
```

- Pipeline create:

```
pipeline RX period 10 offset_port_id 0           (Soft NIC rx-path pipeline)
pipeline TX period 10 offset_port_id 0           (Soft NIC tx-path pipeline)
```

- Pipeline input/output port create

```
pipeline RX port in bsz 32 link LINK rxq 0      (Soft NIC rx pipeline input␣
↪port)
pipeline RX port out bsz 32 swq RXQ0            (Soft NIC rx pipeline output␣
↪port)
pipeline TX port in bsz 32 swq TXQ0             (Soft NIC tx pipeline input␣
```

(continues on next page)

(continued from previous page)

```
↪port)
pipeline TX port out bsz 32 link LINK txq 0     (Soft NIC tx pipeline output␣
↪port)
```

- Pipeline table create

```
pipeline RX table match stub             (Soft NIC rx pipeline match-action␣
↪table)
pipeline TX table match stub             (Soft NIC tx pipeline match-action␣
↪table)
```

- Pipeline input port connection with table

```
pipeline RX port in 0 table 0          (Soft NIC rx pipeline input port 0␣
↪connection with table 0)
pipeline TX port in 0 table 0          (Soft NIC tx pipeline input port 0␣
↪connection with table 0)
```

- Pipeline table match-action rules add

```
pipeline RX table 0 rule add match default action fwd port 0        (Soft NIC␣
↪rx pipeline table 0 rule)
pipeline TX table 0 rule add match default action fwd port 0        (Soft NIC␣
↪tx pipeline table 0 rule)
```

- Enable pipeline on CPU thread

```
thread 1 pipeline RX enable        (Soft NIC rx pipeline enable on cpu thread␣
↪id 1)
thread 1 pipeline TX enable        (Soft NIC tx pipeline enable on cpu thread␣
↪id 1)
```

9.46.7 QoS API Support:

SoftNIC PMD implements ethdev traffic management APIs `rte_tm.h` that allow building and committing traffic manager hierarchy, configuring hierarchy nodes of the Quality of Service (QoS) scheduler supported by DPDK librte_sched library. Furthermore, APIs for run-time update to the traffic manager hierarchy are supported by PMD.

SoftNIC PMD also implements ethdev traffic metering and policing APIs `rte_mtr.h` that enables metering and marking of the packets with the appropriate color (green, yellow or red), according to the traffic metering algorithm. For the meter output color, policer actions like *keep the packet color same*, *change the packet color* or *drop the packet* can be configured.

Note: The SoftNIC does not support the meter object shared by several flows, thus only supports creating meter object private to the flow. Once meter object is successfully created, it can be linked to the specific flow by specifying the `meter` flow action in the flow rule.

9.46.8 Flow API support:

The SoftNIC PMD implements ethdev flow APIs `rte_flow.h` that allow validating flow rules, adding flow rules to the SoftNIC pipeline as table rules, deleting and querying the flow rules. The PMD provides new cli command for creating the flow group and their mapping to the SoftNIC pipeline and table. This cli should be configured as part of firmware file.

```
flowapi map group <group_id> ingress | egress pipeline <pipeline_name> \
    table <table_id>
```

From the flow attributes of the flow, PMD uses the group id to get the mapped pipeline and table. PMD supports number of flow actions such as `JMP, QUEUE, RSS, DROP, COUNT, METER, VXLAN` etc.

Note: The flow must have one terminating actions i.e. `JMP or RSS or QUEUE or DROP`. For the count and drop actions the underlying PMD doesn't support the functionality yet. So it is not recommended for use.

The flow API can be tested with the help of testpmd application. The SoftNIC firmware specifies CLI commands for port configuration, pipeline creation, action profile creation and table creation. Once application gets initialized, the flow rules can be added through the testpmd CLI. The PMD will translate the flow rules to the SoftNIC pipeline tables rules.

Example:

Example demonstrates the flow queue action using the SoftNIC firmware and testpmd commands.

- Prepare SoftNIC firmware

```
link LINK0 dev 0000:83:00.0
link LINK1 dev 0000:81:00.0
pipeline RX period 10 offset_port_id 0
pipeline RX port in bsz 32 link LINK0 rxq 0
pipeline RX port in bsz 32 link LINK1 rxq 0
pipeline RX port out bsz 32 swq RXQ0
pipeline RX port out bsz 32 swq RXQ1
table action profile AP0 ipv4 offset 278 fwd
pipeline RX table match hash ext key 16 mask
    00FF0000FFFFFFFFFFFFFFFFFFFFFFFF \
    offset 278 buckets 16K size 65K action AP0
pipeline RX port in 0 table 0
pipeline RX port in 1 table 0
flowapi map group 0 ingress pipeline RX table 0
pipeline TX period 10 offset_port_id 0
pipeline TX port in bsz 32 swq TXQ0
pipeline TX port in bsz 32 swq TXQ1
pipeline TX port out bsz 32 link LINK0 txq 0
pipeline TX port out bsz 32 link LINK1 txq 0
pipeline TX table match hash ext key 16 mask
    00FF0000FFFFFFFFFFFFFFFFFFFFFFFF \
    offset 278 buckets 16K size 65K action AP0
pipeline TX port in 0 table 0
pipeline TX port in 1 table 0
pipeline TX table 0 rule add match hash ipv4_5tuple
    1.10.11.12 2.20.21.22 100 200 6 action fwd port 0
pipeline TX table 0 rule add match hash ipv4_5tuple
```

(continues on next page)

(continued from previous page)

```
    1.10.11.13 2.20.21.23 100 200 6 action fwd port 1
thread 25 pipeline RX enable
thread 25 pipeline TX enable
```

- Run testpmd:

```
./x86_64-native-linux-gcc/app/testpmd -l 23-25  -n 4 \
                            --vdev 'net_softnic0, \
                            firmware=./drivers/net/softnic/ \
                                firmware.cli, \
                            cpu_id=1,conn_port=8086' -- \
                            -i --forward-mode=softnic --rxq=2, \
                            --txq=2, --disable-rss --portmask=0x4
```

- Configure flow rules on softnic:

```
flow create 2 group 0 ingress pattern eth / ipv4 proto mask 255 src \
    mask 255.255.255.255 dst mask  255.255.255.255 src spec
    1.10.11.12 dst spec 2.20.21.22 proto spec 6 / tcp src mask 65535 \
    dst mask 65535 src spec 100 dst spec 200 / end actions queue \
    index 0 / end
flow create 2 group 0 ingress pattern eth / ipv4 proto mask 255 src \
    mask 255.255.255.255 dst mask  255.255.255.255 src spec 1.10.11.13 \
    dst spec 2.20.21.23 proto spec 6 / tcp src mask 65535 dst mask \
    65535 src spec 100 dst spec 200 / end actions queue index 1 / end
```

9.47 SZEDATA2 poll mode driver library

The SZEDATA2 poll mode driver library implements support for the Netcope FPGA Boards (**NFB-40G2, NFB-100G2, NFB-200G2QL**) and Silicom **FB2CGG3** card, FPGA-based programmable NICs. The SZEDATA2 PMD uses interface provided by the libsze2 library to communicate with the NFB cards over the sze2 layer.

More information about the NFB cards and used technology (Netcope Development Kit) can be found on the Netcope Technologies website.

Note: This driver has external dependencies. Therefore it is disabled in default configuration files. It can be enabled by setting `CONFIG_RTE_LIBRTE_PMD_SZEDATA2=y` and recompiling.

Note: Currently the driver is supported only on x86_64 architectures. Only x86_64 versions of the external libraries are provided.

9.47.1 Prerequisites

This PMD requires kernel modules which are responsible for initialization and allocation of resources needed for sze2 layer function. Communication between PMD and kernel modules is mediated by libsze2 library. These kernel modules and library are not part of DPDK and must be installed separately:

- **libsze2 library**

  The library provides API for initialization of sze2 transfers, receiving and transmitting data segments.

- **Kernel modules**

  - combo6core
  - combov3
  - szedata2
  - szedata2_cv3 or szedata2_cv3_fdt

  Kernel modules manage initialization of hardware, allocation and sharing of resources for user space applications.

Information about getting the dependencies can be found here.

Versions of the packages

The minimum version of the provided packages:

- for DPDK from 18.05: **4.4.1**
- for DPDK up to 18.02 (including): **3.0.5**

9.47.2 Configuration

These configuration options can be modified before compilation in the `.config` file:

- `CONFIG_RTE_LIBRTE_PMD_SZEDATA2` default value: **n**

  Value **y** enables compilation of szedata2 PMD.

9.47.3 Using the SZEDATA2 PMD

From DPDK version 16.04 the type of SZEDATA2 PMD is changed to PMD_PDEV. SZEDATA2 device is automatically recognized during EAL initialization. No special command line options are needed.

Kernel modules have to be loaded before running the DPDK application.

9.47.4 NFB card architecture

The NFB cards are multi-port multi-queue cards, where (generally) data from any Ethernet port may be sent to any queue. They were historically represented in DPDK as a single port.

However, the new NFB-200G2QL card employs an add-on cable which allows to connect it to two physical PCI-E slots at the same time (see the diagram below). This is done to allow 200 Gbps of traffic to be transferred through the PCI-E bus (note that a single PCI-E 3.0 x16 slot provides only 125 Gbps theoretical throughput).

Since each slot may be connected to a different CPU and therefore to a different NUMA node, the card is represented as two ports in DPDK (each with half of the queues), which allows DPDK to work with data from the individual queues on the right NUMA node.

Fig. 9.8: NFB-200G2QL high-level diagram

9.47.5 Limitations

The SZEDATA2 PMD does not support operations related to Ethernet ports (link_up, link_down, set_mac_address, etc.).

NFB cards employ multiple Ethernet ports. Until now, Ethernet port-related operations were performed on all of them (since the whole card was represented as a single port). With NFB-200G2QL card, this is no longer viable (see above).

Since there is no fixed mapping between the queues and Ethernet ports, and since a single card can be represented as two ports in DPDK, there is no way of telling which (if any) physical ports should be associated with individual ports in DPDK.

9.47.6 Example of usage

Read packets from 0. and 1. receive channel and write them to 0. and 1. transmit channel:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 2 \
-- --port-topology=chained --rxq=2 --txq=2 --nb-cores=2 -i -a
```

Example output:

```
[...]
EAL: PCI device 0000:06:00.0 on NUMA socket -1
EAL:   probe driver: 1b26:c1c1 rte_szedata2_pmd
PMD: Initializing szedata2 device (0000:06:00.0)
PMD: SZEDATA2 path: /dev/szedataII0
PMD: Available DMA channels RX: 8 TX: 8
PMD: resource0 phys_addr = 0xe8000000 len = 134217728 virt addr = 7f48f8000000
PMD: szedata2 device (0000:06:00.0) successfully initialized
Interactive-mode selected
Auto-start selected
Configuring Port 0 (socket 0)
Port 0: 00:11:17:00:00:00
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
Start automatic packet forwarding
```

(continues on next page)

(continued from previous page)

```
  io packet forwarding - CRC stripping disabled - packets/burst=32
  nb forwarding cores=2 - nb forwarding ports=1
  RX queues=2 - RX desc=128 - RX free threshold=0
  RX threshold registers: pthresh=0 hthresh=0 wthresh=0
  TX queues=2 - TX desc=512 - TX free threshold=0
  TX threshold registers: pthresh=0 hthresh=0 wthresh=0
  TX RS bit threshold=0 - TXQ flags=0x0
testpmd>
```

9.48 Tun|Tap Poll Mode Driver

The `rte_eth_tap.c` PMD creates a device using TAP interfaces on the local host. The PMD allows for DPDK and the host to communicate using a raw device interface on the host and in the DPDK application.

The device created is a TAP device, which sends/receives packet in a raw format with a L2 header. The usage for a TAP PMD is for connectivity to the local host using a TAP interface. When the TAP PMD is initialized it will create a number of tap devices in the host accessed via `ifconfig -a` or `ip` command. The commands can be used to assign and query the virtual like device.

These TAP interfaces can be used with Wireshark or tcpdump or Pktgen-DPDK along with being able to be used as a network connection to the DPDK application. The method enable one or more interfaces is to use the `--vdev=net_tap0` option on the DPDK application command line. Each `--vdev=net_tap1` option given will create an interface named dtap0, dtap1, and so on.

The interface name can be changed by adding the `iface=foo0`, for example:

```
--vdev=net_tap0,iface=foo0 --vdev=net_tap1,iface=foo1, ...
```

Normally the PMD will generate a random MAC address, but when testing or with a static configuration the developer may need a fixed MAC address style. Using the option `mac=fixed` you can create a fixed known MAC address:

```
--vdev=net_tap0,mac=fixed
```

The MAC address will have a fixed value with the last octet incrementing by one for each interface string containing `mac=fixed`. The MAC address is formatted as 00:'d':'t':'a':'p':[00-FF]. Convert the characters to hex and you get the actual MAC address: `00:64:74:61:70:[00-FF]`.

–vdev=net_tap0,mac="00:64:74:61:70:11"

The MAC address will have a user value passed as string. The MAC address is in format with delimiter `:`. The string is byte converted to hex and you get the actual MAC address: `00:64:74:61:70:11`.

It is possible to specify a remote netdevice to capture packets from by adding `remote=foo1`, for example:

```
--vdev=net_tap,iface=tap0,remote=foo1
```

If a `remote` is set, the tap MAC address will be set to match the remote one just after netdevice creation. Using TC rules, traffic from the remote netdevice will be redirected to the tap. If the tap is in promiscuous mode, then all packets will be redirected. In allmulti mode, all multicast packets will be redirected.

Using the remote feature is especially useful for capturing traffic from a netdevice that has no support in the DPDK. It is possible to add explicit rte_flow rules on the tap PMD to capture specific traffic (see next section for examples).

After the DPDK application is started you can send and receive packets on the interface using the standard rx_burst/tx_burst APIs in DPDK. From the host point of view you can use any host tool like tcpdump, Wireshark, ping, Pktgen and others to communicate with the DPDK application. The DPDK application may not understand network protocols like IPv4/6, UDP or TCP unless the application has been written to understand these protocols.

If you need the interface as a real network interface meaning running and has a valid IP address then you can do this with the following commands:

```
sudo ip link set dtap0 up; sudo ip addr add 192.168.0.250/24 dev dtap0
sudo ip link set dtap1 up; sudo ip addr add 192.168.1.250/24 dev dtap1
```

Please change the IP addresses as you see fit.

If routing is enabled on the host you can also communicate with the DPDK App over the internet via a standard socket layer application as long as you account for the protocol handling in the application.

If you have a Network Stack in your DPDK application or something like it you can utilize that stack to handle the network protocols. Plus you would be able to address the interface using an IP address assigned to the internal interface.

The TUN PMD allows user to create a TUN device on host. The PMD allows user to transmit and receive packets via DPDK API calls with L3 header and payload. The devices in host can be accessed via `ifconfig` or `ip` command. TUN interfaces are passed to DPDK `rte_eal_init` arguments as `--vdev=net_tunX`, where X stands for unique id, example:

```
--vdev=net_tun0 --vdev=net_tun1,iface=foo1, ...
```

Unlike TAP PMD, TUN PMD does not support user arguments as `MAC` or `remote` user options. Default interface name is `dtunX`, where X stands for unique id.

9.48.1 Flow API support

The tap PMD supports major flow API pattern items and actions, when running on linux kernels above 4.2 ("Flower" classifier required). The kernel support can be checked with this command:

```
zcat /proc/config.gz | ( grep 'CLS_FLOWER=' || echo 'not supported' ) |
tee -a /dev/stderr | grep -q '=m' &&
lsmod | ( grep cls_flower || echo 'try modprobe cls_flower' )
```

Supported items:

- eth: src and dst (with variable masks), and eth_type (0xffff mask).
- vlan: vid, pcp, but not eid. (requires kernel 4.9)
- ipv4/6: src and dst (with variable masks), and ip_proto (0xffff mask).
- udp/tcp: src and dst port (0xffff) mask.

Supported actions:

- DROP
- QUEUE
- PASSTHRU
- RSS (requires kernel 4.9)

It is generally not possible to provide a "last" item. However, if the "last" item, once masked, is identical to the masked spec, then it is supported.

Only IPv4/6 and MAC addresses can use a variable mask. All other items need a full mask (exact match).

As rules are translated to TC, it is possible to show them with something like:

```
tc -s filter show dev tap1 parent 1:
```

Examples of testpmd flow rules

Drop packets for destination IP 192.0.2.1:

```
testpmd> flow create 0 priority 1 ingress pattern eth / ipv4 dst is 192.0.2.1 \
         / end actions drop / end
```

Ensure packets from a given MAC address are received on a queue 2:

```
testpmd> flow create 0 priority 2 ingress pattern eth src is 06:05:04:03:02:01 \
         / end actions queue index 2 / end
```

Drop UDP packets in vlan 3:

```
testpmd> flow create 0 priority 3 ingress pattern eth / vlan vid is 3 / \
         ipv4 proto is 17 / end actions drop / end
```

Distribute IPv4 TCP packets using RSS to a given MAC address over queues 0-3:

```
testpmd> flow create 0 priority 4 ingress pattern eth dst is 0a:0b:0c:0d:0e:0f \
         / ipv4 / tcp / end actions rss queues 0 1 2 3 end / end
```

9.48.2 Multi-process sharing

It is possible to attach an existing TAP device in a secondary process, by declaring it as a vdev with the same name as in the primary process, and without any parameter.

The port attached in a secondary process will give access to the statistics and the queues. Therefore it can be used for monitoring or Rx/Tx processing.

The IPC synchronization of Rx/Tx queues is currently limited:

- Maximum 8 queues shared
- Synchronized on probing, but not on later port update

9.48.3 Example

The following is a simple example of using the TAP PMD with the Pktgen packet generator. It requires that the `socat` utility is installed on the test system.

Build DPDK, then pull down Pktgen and build pktgen using the DPDK SDK/Target used to build the dpdk you pulled down.

Run pktgen from the pktgen directory in a terminal with a commandline like the following:

```
sudo ./app/app/x86_64-native-linux-gcc/app/pktgen -l 1-5 -n 4        \
 --proc-type auto --log-level debug --socket-mem 512,512 --file-prefix pg   \
 --vdev=net_tap0 --vdev=net_tap1 -b 05:00.0 -b 05:00.1                  \
 -b 04:00.0 -b 04:00.1 -b 04:00.2 -b 04:00.3                            \
 -b 81:00.0 -b 81:00.1 -b 81:00.2 -b 81:00.3                            \
 -b 82:00.0 -b 83:00.0 -- -T -P -m [2:3].0 -m [4:5].1                   \
 -f themes/black-yellow.theme
```

Verify with `ifconfig -a` command in a different xterm window, should have a `dtap0` and `dtap1` interfaces created.

Next set the links for the two interfaces to up via the commands below:

```
sudo ip link set dtap0 up; sudo ip addr add 192.168.0.250/24 dev dtap0
sudo ip link set dtap1 up; sudo ip addr add 192.168.1.250/24 dev dtap1
```

Then use socat to create a loopback for the two interfaces:

```
sudo socat interface:dtap0 interface:dtap1
```

Then on the Pktgen command line interface you can start sending packets using the commands `start 0` and `start 1` or you can start both at the same time with `start all`. The command `str` is an alias for `start all` and `stp` is an alias for `stop all`.

While running you should see the 64 byte counters increasing to verify the traffic is being looped back. You can use `set all size XXX` to change the size of the packets after you stop the traffic. Use pktgen `help` command to see a list of all commands. You can also use the `-f` option to load commands at startup in command line or Lua script in pktgen.

9.48.4 RSS specifics

Packet distribution in TAP is done by the kernel which has a default distribution. This feature is adding RSS distribution based on eBPF code. The default eBPF code calculates RSS hash based on Toeplitz algorithm for a fixed RSS key. It is calculated on fixed packet offsets. For IPv4 and IPv6 it is calculated over src/dst addresses (8 or 32 bytes for IPv4 or IPv6 respectively) and src/dst TCP/UDP ports (4 bytes).

The RSS algorithm is written in file `tap_bpf_program.c` which does not take part in TAP PMD compilation. Instead this file is compiled in advance to eBPF object file. The eBPF object file is then parsed and translated into eBPF byte code in the format of C arrays of eBPF instructions. The C array of eBPF instructions is part of TAP PMD tree and is taking part in TAP PMD compilation. At run time the C arrays are uploaded to the kernel via BPF system calls and the RSS hash is calculated by the kernel.

It is possible to support different RSS hash algorithms by updating file `tap_bpf_program.c` In order to add a new RSS hash algorithm follow these steps:

1. Write the new RSS implementation in file `tap_bpf_program.c`

BPF programs which are uploaded to the kernel correspond to C functions under different ELF sections.

2. Install `LLVM` library and `clang` compiler versions 3.7 and above
3. Compile `tap_bpf_program.c` via `LLVM` into an object file:

```
clang -O2 -emit-llvm -c tap_bpf_program.c -o - | llc -march=bpf \
-filetype=obj -o <tap_bpf_program.o>
```

4. Use a tool that receives two parameters: an eBPF object file and a section name, and prints out the section as a C array of eBPF instructions. Embed the C array in your TAP PMD tree.

The C arrays are uploaded to the kernel using BPF system calls.

`tc` (traffic control) is a well known user space utility program used to configure the Linux kernel packet scheduler. It is usually packaged as part of the `iproute2` package. Since commit 11c39b5e9 ("tc: add eBPF support to f_bpf") `tc` can be used to uploads eBPF code to the kernel and can be patched in order to print the C arrays of eBPF instructions just before calling the BPF system call. Please refer to `iproute2` package file `lib/bpf.c` function `bpf_prog_load()`.

An example utility for eBPF instruction generation in the format of C arrays will be added in next releases

TAP reports on supported RSS functions as part of dev_infos_get callback: `ETH_RSS_IP`, `ETH_RSS_UDP` and `ETH_RSS_TCP`. **Known limitation:** TAP supports all of the above hash functions together and not in partial combinations.

9.48.5 Systems supporting flow API

- "tc flower" classifier requires linux kernel above 4.2
- eBPF/RSS requires linux kernel above 4.9

RH7.3	No flow rule support
RH7.4	No RSS action support
RH7.5	No RSS action support
SLES 15, kernel 4.12	No limitation
Azure Ubuntu 16.04, kernel 4.13	No limitation

9.49 ThunderX NICVF Poll Mode Driver

The ThunderX NICVF PMD (**librte_pmd_thunderx_nicvf**) provides poll mode driver support for the inbuilt NIC found in the **Cavium ThunderX** SoC family as well as their virtual functions (VF) in SR-IOV context.

More information can be found at Cavium, Inc Official Website.

9.49.1 Features

Features of the ThunderX PMD are:

- Multiple queues for TX and RX
- Receive Side Scaling (RSS)
- Packet type information
- Checksum offload
- Promiscuous mode
- Multicast mode
- Port hardware statistics

- Jumbo frames
- Link state information
- Setting up link state.
- Scattered and gather for TX and RX
- VLAN stripping
- SR-IOV VF
- NUMA support
- Multi queue set support (up to 96 queues (12 queue sets)) per port
- Skip data bytes

9.49.2 Supported ThunderX SoCs

- CN88xx
- CN81xx
- CN83xx

9.49.3 Prerequisites

- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

9.49.4 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_PMD` (default `y`)

  Toggle compilation of the `librte_pmd_thunderx_nicvf` driver.

- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_DEBUG_RX` (default `n`)

  Toggle asserts of receive fast path.

- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_DEBUG_TX` (default `n`)

  Toggle asserts of transmit fast path.

9.49.5 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

To compile the ThunderX NICVF PMD for Linux arm64 gcc, use arm64-thunderx-linux-gcc as target.

9.49.6 Linux

SR-IOV: Prerequisites and sample Application Notes

Current ThunderX NIC PF/VF kernel modules maps each physical Ethernet port automatically to virtual function (VF) and presented them as PCIe-like SR-IOV device. This section provides instructions to configure SR-IOV with Linux OS.

1. Verify PF devices capabilities using `lspci`:

```
lspci -vvv
```

Example output:

```
0002:01:00.0 Ethernet controller: Cavium Networks Device a01e (rev 01)
...
Capabilities: [100 v1] Alternative Routing-ID Interpretation (ARI)
...
Capabilities: [180 v1] Single Root I/O Virtualization (SR-IOV)
...
Kernel driver in use: thunder-nic
...
```

Note: Unless `thunder-nic` driver is in use make sure your kernel config includes `CONFIG_THUNDER_NIC_PF` setting.

2. Verify VF devices capabilities and drivers using `lspci`:

```
lspci -vvv
```

Example output:

```
0002:01:00.1 Ethernet controller: Cavium Networks Device 0011 (rev 01)
...
Capabilities: [100 v1] Alternative Routing-ID Interpretation (ARI)
...
Kernel driver in use: thunder-nicvf
...

0002:01:00.2 Ethernet controller: Cavium Networks Device 0011 (rev 01)
...
Capabilities: [100 v1] Alternative Routing-ID Interpretation (ARI)
...
Kernel driver in use: thunder-nicvf
...
```

Note: Unless `thunder-nicvf` driver is in use make sure your kernel config includes `CONFIG_THUNDER_NIC_VF` setting.

3. Pass VF device to VM context (PCIe Passthrough):

   The VF devices may be passed through to the guest VM using qemu or virt-manager or virsh etc.

   Example qemu guest launch command:

```
sudo qemu-system-aarch64 -name vm1 \
-machine virt,gic_version=3,accel=kvm,usb=off \
-cpu host -m 4096 \
-smp 4,sockets=1,cores=8,threads=1 \
-nographic -nodefaults \
-kernel <kernel image> \
-append "root=/dev/vda console=ttyAMA0 rw hugepagesz=512M hugepages=3" \
-device vfio-pci,host=0002:01:00.1 \
-drive file=<rootfs.ext3>,if=none,id=disk1,format=raw  \
-device virtio-blk-device,scsi=off,drive=disk1,id=virtio-disk1,bootindex=1 \
-netdev tap,id=net0,ifname=tap0,script=/etc/qemu-ifup_thunder \
-device virtio-net-device,netdev=net0 \
-serial stdio \
-mem-path /dev/huge
```

4. Enable **VFIO-NOIOMMU** mode (optional):

```
echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode
```

Note: **VFIO-NOIOMMU** is required only when running in VM context and should not be enabled otherwise.

5. Running testpmd:

   Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

   Example output:

```
./arm64-thunderx-linux-gcc/app/testpmd -l 0-3 -n 4 -w 0002:01:00.2 \
  -- -i --no-flush-rx \
  --port-topology=loop

...

PMD: rte_nicvf_pmd_init(): librte_pmd_thunderx nicvf version 1.0

...
EAL:   probe driver: 177d:11 rte_nicvf_pmd
EAL:   using IOMMU type 1 (Type 1)
EAL:   PCI memory mapped at 0x3ffade50000
EAL: Trying to map BAR 4 that contains the MSI-X table.
     Trying offsets: 0x40000000000:0x0000, 0x10000:0x1f0000
EAL:   PCI memory mapped at 0x3ffadc60000
PMD: nicvf_eth_dev_init(): nicvf: device (177d:11) 2:1:0:2
PMD: nicvf_eth_dev_init(): node=0 vf=1 mode=tns-bypass sqs=false
     loopback_supported=true
PMD: nicvf_eth_dev_init(): Port 0 (177d:11) mac=a6:c6:d9:17:78:01
Interactive-mode selected
Configuring Port 0 (socket 0)
...

PMD: nicvf_dev_configure(): Configured ethdev port0 hwcap=0x0
```

(continues on next page)

(continued from previous page)

```
Port 0: A6:C6:D9:17:78:01
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

Multiple Queue Set per DPDK port configuration

There are two types of VFs:

- Primary VF
- Secondary VF

Each port consists of a primary VF and n secondary VF(s). Each VF provides 8 Tx/Rx queues to a port. When a given port is configured to use more than 8 queues, it requires one (or more) secondary VF. Each secondary VF adds 8 additional queues to the queue set.

During PMD driver initialization, the primary VF's are enumerated by checking the specific flag (see sqs message in DPDK boot log - sqs indicates secondary queue set). They are at the beginning of VF list (the remain ones are secondary VF's).

The primary VFs are used as master queue sets. Secondary VFs provide additional queue sets for primary ones. If a port is configured for more then 8 queues than it will request for additional queues from secondary VFs.

Secondary VFs cannot be shared between primary VFs.

Primary VFs are present on the beginning of the 'Network devices using kernel driver' list, secondary VFs are on the remaining on the remaining part of the list.

Note: The VNIC driver in the multiqueue setup works differently than other drivers like *ixgbe*. We need to bind separately each specific queue set device with the `usertools/dpdk-devbind.py` utility.

Note: Depending on the hardware used, the kernel driver sets a threshold `vf_id`. VFs that try to attached with an id below or equal to this boundary are considered primary VFs. VFs that try to attach with an id above this boundary are considered secondary VFs.

LBK HW Access

Loopback HW Unit (LBK) receives packets from NIC-RX and sends packets back to NIC-TX. The loopback block has N channels and contains data buffering that is shared across all channels. Four primary VFs are reserved as loopback ports.

Example device binding

If a system has three interfaces, a total of 18 VF devices will be created on a non-NUMA machine.

Note: NUMA systems have 12 VFs per port and non-NUMA 6 VFs per port.

```
# usertools/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
============================================
<none>

Network devices using kernel driver
===================================
0000:01:10.0 'THUNDERX BGX (Common Ethernet Interface) a026' if= drv=thunder-BGX␣
↪unused=vfio-pci
0000:01:10.1 'THUNDERX BGX (Common Ethernet Interface) a026' if= drv=thunder-BGX␣
↪unused=vfio-pci
0001:01:00.0 'THUNDERX Network Interface Controller a01e' if= drv=thunder-nic␣
↪unused=vfio-pci
0001:01:00.1 'Device a034' if=eth0 drv=thunder-nicvf unused=vfio-pci
0001:01:00.2 'Device a034' if=eth1 drv=thunder-nicvf unused=vfio-pci
0001:01:00.3 'Device a034' if=eth2 drv=thunder-nicvf unused=vfio-pci
0001:01:00.4 'Device a034' if=eth3 drv=thunder-nicvf unused=vfio-pci
0001:01:00.5 'Device a034' if=eth4 drv=thunder-nicvf unused=vfio-pci
0001:01:00.6 'Device a034' if=lbk0 drv=thunder-nicvf unused=vfio-pci
0001:01:00.7 'Device a034' if=lbk1 drv=thunder-nicvf unused=vfio-pci
0001:01:01.0 'Device a034' if=lbk2 drv=thunder-nicvf unused=vfio-pci
0001:01:01.1 'Device a034' if=lbk3 drv=thunder-nicvf unused=vfio-pci
0001:01:01.2 'Device a034' if= drv=thunder-nicvf unused=vfio-pci
0001:01:01.3 'Device a034' if= drv=thunder-nicvf unused=vfio-pci
0001:01:01.4 'Device a034' if= drv=thunder-nicvf unused=vfio-pci
0001:01:01.5 'Device a034' if= drv=thunder-nicvf unused=vfio-pci
0001:01:01.6 'Device a034' if= drv=thunder-nicvf unused=vfio-pci
0001:01:01.7 'Device a034' if= drv=thunder-nicvf unused=vfio-pci
0001:01:02.0 'Device a034' if= drv=thunder-nicvf unused=vfio-pci
0001:01:02.1 'Device a034' if= drv=thunder-nicvf unused=vfio-pci
0001:01:02.2 'Device a034' if= drv=thunder-nicvf unused=vfio-pci

Other network devices
=====================
0002:00:03.0 'Device a01f' unused=vfio-pci,uio_pci_generic
```

Note: Here total no of primary VFs = 5 (variable, depends on no of ethernet ports present) + 4 (fixed, loopback ports). Ethernet ports are indicated as *if=eth0* while loopback ports as *if=lbk0*.

We want to bind two physical interfaces with 24 queues each device, we attach two primary VFs and four secondary VFs. In our example we choose two 10G interfaces eth1 (0002:01:00.2) and eth2 (0002:01:00.3). We will choose four secondary queue sets from the ending of the list (0001:01:01.2-0002:01:02.2).

1. Bind two primary VFs to the `vfio-pci` driver:

```
usertools/dpdk-devbind.py -b vfio-pci 0002:01:00.2
usertools/dpdk-devbind.py -b vfio-pci 0002:01:00.3
```

2. Bind four primary VFs to the `vfio-pci` driver:

```
usertools/dpdk-devbind.py -b vfio-pci 0002:01:01.7
usertools/dpdk-devbind.py -b vfio-pci 0002:01:02.0
usertools/dpdk-devbind.py -b vfio-pci 0002:01:02.1
usertools/dpdk-devbind.py -b vfio-pci 0002:01:02.2
```

The nicvf thunderx driver will make use of attached secondary VFs automatically during the interface configuration stage.

Thunder-nic VF's

Use sysfs to distinguish thunder-nic primary VFs and secondary VFs.

```
ls -l /sys/bus/pci/drivers/thunder-nic/
total 0
drwxr-xr-x  2 root root     0 Jan 22 11:19 ./
drwxr-xr-x 86 root root     0 Jan 22 11:07 ../
lrwxrwxrwx  1 root root     0 Jan 22 11:19 0001:01:00.0 -> '../../../../devices/platform/
↪soc@0/849000000000.pci/pci0001:00/0001:00:10.0/0001:01:00.0'/
```

```
cat /sys/bus/pci/drivers/thunder-nic/0001\:01\:00.0/sriov_sqs_assignment
12
0 0001:01:00.1 vfio-pci +: 12 13
1 0001:01:00.2 thunder-nicvf -:
2 0001:01:00.3 thunder-nicvf -:
3 0001:01:00.4 thunder-nicvf -:
4 0001:01:00.5 thunder-nicvf -:
5 0001:01:00.6 thunder-nicvf -:
6 0001:01:00.7 thunder-nicvf -:
7 0001:01:01.0 thunder-nicvf -:
8 0001:01:01.1 thunder-nicvf -:
9 0001:01:01.2 thunder-nicvf -:
10 0001:01:01.3 thunder-nicvf -:
11 0001:01:01.4 thunder-nicvf -:
12 0001:01:01.5 vfio-pci: 0
13 0001:01:01.6 vfio-pci: 0
14 0001:01:01.7 thunder-nicvf: 255
15 0001:01:02.0 thunder-nicvf: 255
16 0001:01:02.1 thunder-nicvf: 255
17 0001:01:02.2 thunder-nicvf: 255
18 0001:01:02.3 thunder-nicvf: 255
19 0001:01:02.4 thunder-nicvf: 255
20 0001:01:02.5 thunder-nicvf: 255
21 0001:01:02.6 thunder-nicvf: 255
22 0001:01:02.7 thunder-nicvf: 255
23 0001:01:03.0 thunder-nicvf: 255
24 0001:01:03.1 thunder-nicvf: 255
25 0001:01:03.2 thunder-nicvf: 255
26 0001:01:03.3 thunder-nicvf: 255
27 0001:01:03.4 thunder-nicvf: 255
28 0001:01:03.5 thunder-nicvf: 255
29 0001:01:03.6 thunder-nicvf: 255
30 0001:01:03.7 thunder-nicvf: 255
31 0001:01:04.0 thunder-nicvf: 255
```

Every column that ends with 'thunder-nicvf: number' can be used as secondary VF. In printout above all entres after '14 0001:01:01.7 thunder-nicvf: 255' can be used as secondary VF.

9.49.7 Debugging Options

EAL command option to change log level

```
--log-level=pmd.net.thunderx.driver:info
or
--log-level=pmd.net.thunderx.driver,7
```

9.49.8 Module params

skip_data_bytes

This feature is used to create a hole between HEADROOM and actual data. Size of hole is specified in bytes as module param("skip_data_bytes") to pmd. This scheme is useful when application would like to insert vlan header without disturbing HEADROOM.

Example:

```
-w 0002:01:00.2,skip_data_bytes=8
```

9.49.9 Limitations

CRC stripping

The ThunderX SoC family NICs strip the CRC for every packets coming into the host interface irrespective of the offload configuration.

Maximum packet length

The ThunderX SoC family NICs support a maximum of a 9K jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 9200, frames up to 9200 bytes can still reach the host interface.

Maximum packet segments

The ThunderX SoC family NICs support up to 12 segments per packet when working in scatter/gather mode. So, setting MTU will result with `EINVAL` when the frame size does not fit in the maximum number of segments.

skip_data_bytes

Maximum limit of skip_data_bytes is 128 bytes and number of bytes should be multiple of 8.

9.50 VDEV_NETVSC driver

The VDEV_NETVSC driver (librte_pmd_vdev_netvsc) provides support for NetVSC interfaces and associated SR-IOV virtual function (VF) devices found in Linux virtual machines running on Microsoft Hyper-V (including Azure) platforms.

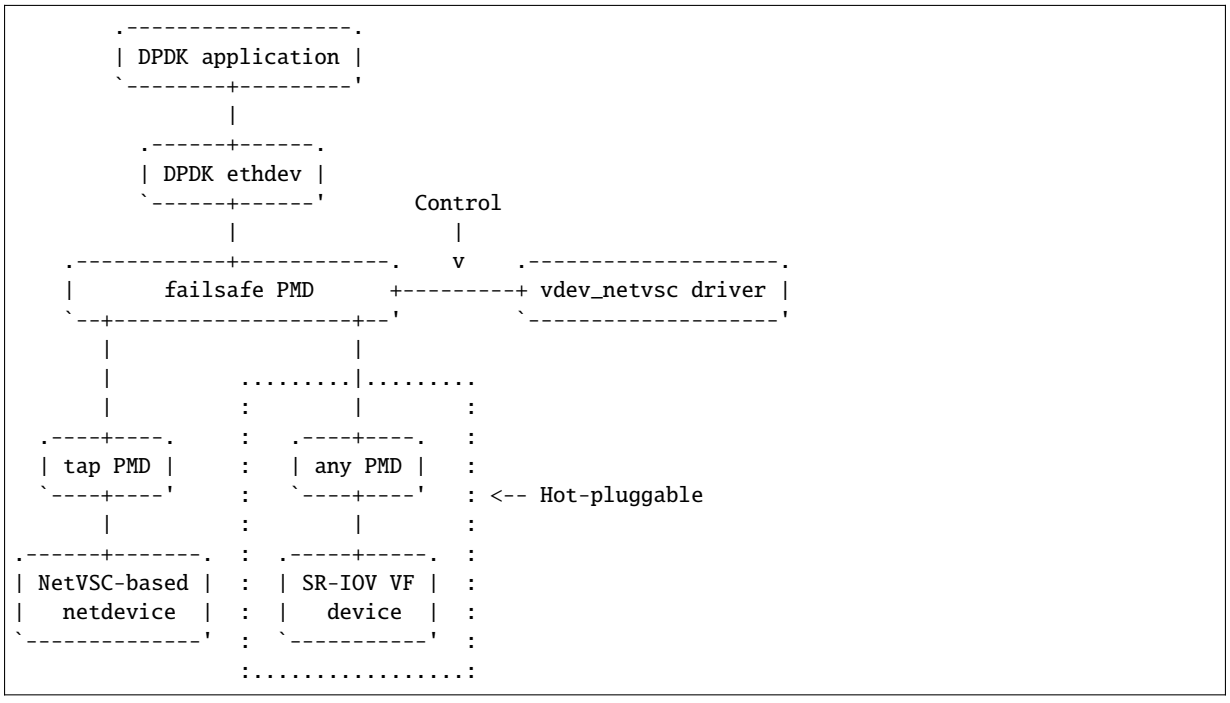
9.50.1 Implementation details

Each instance of this driver effectively needs to drive two devices: the NetVSC interface proper and its SR-IOV VF (referred to as "physical" from this point on) counterpart sharing the same MAC address.

Physical devices are part of the host system and cannot be maintained during VM migration. From a VM standpoint they appear as hot-plug devices that come and go without prior notice.

When the physical device is present, egress and most of the ingress traffic flows through it; only multicasts and other hypervisor control still flow through NetVSC. Otherwise, NetVSC acts as a fallback for all traffic.

To avoid unnecessary code duplication and ensure maximum performance, handling of physical devices is left to their original PMDs; this virtual device driver (also known as *vdev*) manages other PMDs as summarized by the following block diagram:



This driver implementation may be temporary and should be improved or removed either when hot-plug will be fully supported in EAL and bus drivers or when a new NetVSC driver will be integrated.

9.50.2 Build options

- `CONFIG_RTE_LIBRTE_VDEV_NETVSC_PMD` (default `y`)

  Toggle compilation of this driver.

9.50.3 Run-time parameters

This driver is invoked automatically in Hyper-V VM systems unless the user invoked it by command line using `--vdev=net_vdev_netvsc` EAL option.

The following device parameters are supported:

- `iface` [string]

  Provide a specific NetVSC interface (netdevice) name to attach this driver to. Can be provided multiple times for additional instances.

- `mac` [string]

  Same as `iface` except a suitable NetVSC interface is located using its MAC address.

- `force` [int]

  If nonzero, forces the use of specified interfaces even if not detected as NetVSC.

- `ignore` [int]

  If nonzero, ignores the driver running (actually used to disable the auto-detection in Hyper-V VM).

Note: Not specifying either `iface` or `mac` makes this driver attach itself to all unrouted NetVSC interfaces found on the system. Specifying the device makes this driver attach itself to the device regardless the device routes.

9.51 Poll Mode Driver for Emulated Virtio NIC

Virtio is a para-virtualization framework initiated by IBM, and supported by KVM hypervisor. In the Data Plane Development Kit (DPDK), we provide a virtio Poll Mode Driver (PMD) as a software solution, comparing to SRIOV hardware solution, for fast guest VM to guest VM communication and guest VM to host communication.

Vhost is a kernel acceleration module for virtio qemu backend. The DPDK extends kni to support vhost raw socket interface, which enables vhost to directly read/ write packets from/to a physical port. With this enhancement, virtio could achieve quite promising performance.

For basic qemu-KVM installation and other Intel EM poll mode driver in guest VM, please refer to Chapter "Driver for VM Emulated Devices".

In this chapter, we will demonstrate usage of virtio PMD driver with two backends, standard qemu vhost back end and vhost kni back end.

9.51.1 Virtio Implementation in DPDK

For details about the virtio spec, refer to the latest VIRTIO (Virtual I/O) Device Specification.

As a PMD, virtio provides packet reception and transmission callbacks.

In Rx, packets described by the used descriptors in vring are available for virtio to burst out.

In Tx, packets described by the used descriptors in vring are available for virtio to clean. Virtio will enqueue to be transmitted packets into vring, make them available to the device, and then notify the host back end if necessary.

9.51.2 Features and Limitations of virtio PMD

In this release, the virtio PMD driver provides the basic functionality of packet reception and transmission.

- It supports merge-able buffers per packet when receiving packets and scattered buffer per packet when transmitting packets. The packet size supported is from 64 to 1518.
- It supports multicast packets and promiscuous mode.
- The descriptor number for the Rx/Tx queue is hard-coded to be 256 by qemu 2.7 and below. If given a different descriptor number by the upper application, the virtio PMD generates a warning and fall back to the hard-coded value. Rx queue size can be configurable and up to 1024 since qemu 2.8 and above. Rx queue size is 256 by default. Tx queue size is still hard-coded to be 256.
- Features of mac/vlan filter are supported, negotiation with vhost/backend are needed to support them. When backend can't support vlan filter, virtio app on guest should not enable vlan filter in order to make sure the virtio port is configured correctly. E.g. do not specify '–enable-hw-vlan' in testpmd command line. Note that, mac/vlan filter is best effort: unwanted packets could still arrive.
- "RTE_PKTMBUF_HEADROOM" should be defined no less than "sizeof(struct virtio_net_hdr_mrg_rxbuf)", which is 12 bytes when mergeable or "VIRTIO_F_VERSION_1" is set. no less than "sizeof(struct virtio_net_hdr)", which is 10 bytes, when using non-mergeable.
- Virtio does not support runtime configuration.
- Virtio supports Link State interrupt.
- Virtio supports Rx interrupt (so far, only support 1:1 mapping for queue/interrupt).
- Virtio supports software vlan stripping and inserting.
- Virtio supports using port IO to get PCI resource when uio/igb_uio module is not available.

9.51.3 Prerequisites

The following prerequisites apply:

- In the BIOS, turn VT-x and VT-d on
- Linux kernel with KVM module; vhost module loaded and ioeventfd supported. Qemu standard backend without vhost support isn't tested, and probably isn't supported.

9.51.4 Virtio with kni vhost Back End

This section demonstrates kni vhost back end example setup for Phy-VM Communication.

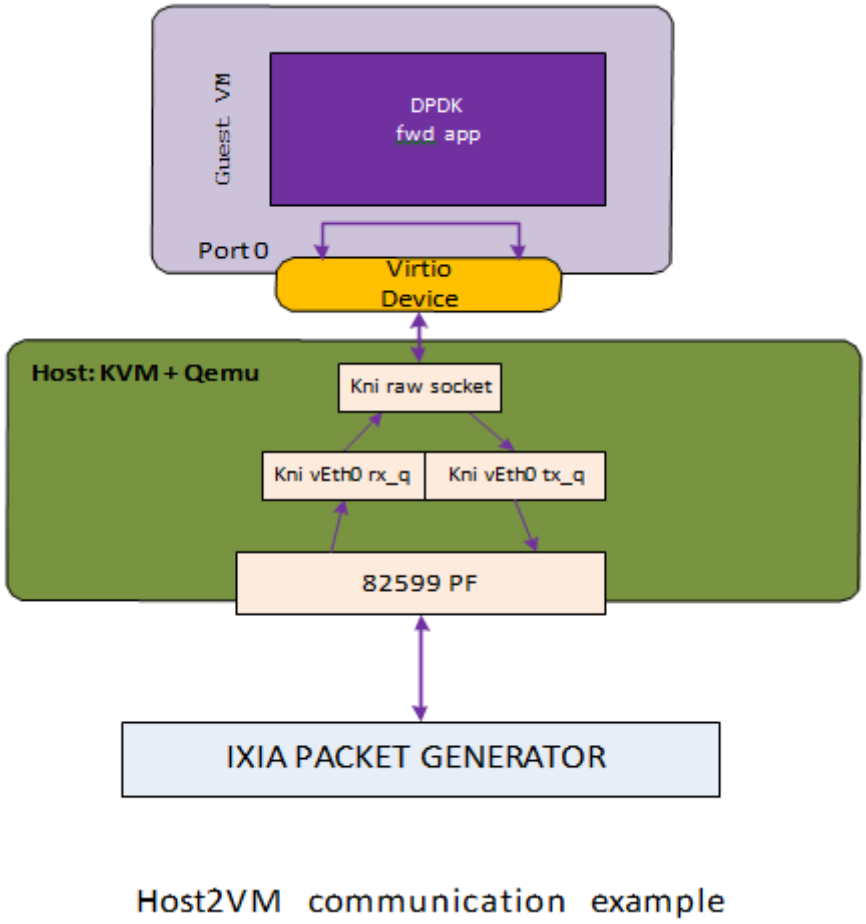


Fig. 9.9: Host2VM Communication Example Using kni vhost Back End

Host2VM communication example

1. Load the kni kernel module:

   ```
   insmod rte_kni.ko
   ```

   Other basic DPDK preparations like hugepage enabling, uio port binding are not listed here. Please refer to the *DPDK Getting Started Guide* for detailed instructions.

2. Launch the kni user application:

   ```
   examples/kni/build/app/kni -l 0-3 -n 4 -- -p 0x1 -P --config="(0,1,3)"
   ```

   This command generates one network device vEth0 for physical port. If specify more physical ports, the generated network device will be vEth1, vEth2, and so on.

   For each physical port, kni creates two user threads. One thread loops to fetch packets from the physical NIC port into the kni receive queue. The other user thread loops to send packets in the kni transmit queue.

   For each physical port, kni also creates a kernel thread that retrieves packets from the kni receive queue, place them onto kni's raw socket's queue and wake up the vhost kernel thread to exchange

packets with the virtio virt queue.

For more details about kni, please refer to *Kernel NIC Interface*.

3. Enable the kni raw socket functionality for the specified physical NIC port, get the generated file descriptor and set it in the qemu command line parameter. Always remember to set ioeventfd_on and vhost_on.

   Example:

```
echo 1 > /sys/class/net/vEth0/sock_en
fd=`cat /sys/class/net/vEth0/sock_fd`
exec qemu-system-x86_64 -enable-kvm -cpu host \
-m 2048 -smp 4 -name dpdk-test1-vm1 \
-drive file=/data/DPDKVMS/dpdk-vm.img \
-netdev tap, fd=$fd,id=mynet_kni, script=no,vhost=on \
-device virtio-net-pci,netdev=mynet_kni,bus=pci.0,addr=0x3,ioeventfd=on \
-vnc:1 -daemonize
```

   In the above example, virtio port 0 in the guest VM will be associated with vEth0, which in turns corresponds to a physical port, which means received packets come from vEth0, and transmitted packets is sent to vEth0.

4. In the guest, bind the virtio device to the uio_pci_generic kernel module and start the forwarding application. When the virtio port in guest bursts Rx, it is getting packets from the raw socket's receive queue. When the virtio port bursts Tx, it is sending packet to the tx_q.

```
modprobe uio
echo 512 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
modprobe uio_pci_generic
python usertools/dpdk-devbind.py -b uio_pci_generic 00:03.0
```

   We use testpmd as the forwarding application in this example.

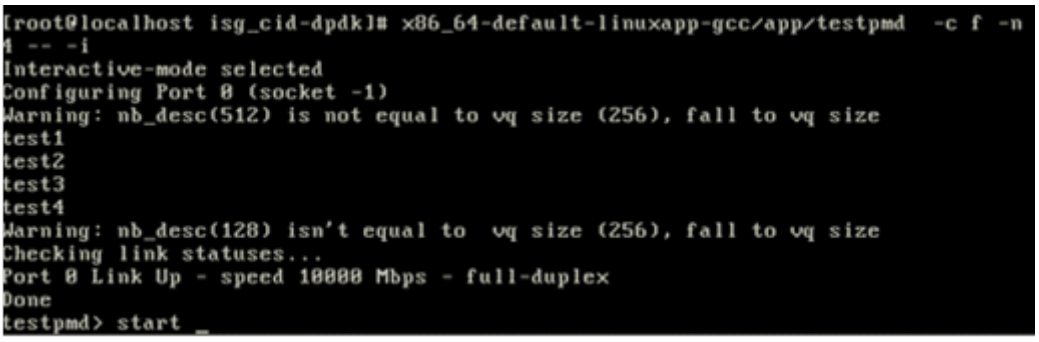


Fig. 9.10: Running testpmd

5. Use IXIA packet generator to inject a packet stream into the KNI physical port.

   The packet reception and transmission flow path is:

   IXIA packet generator->82599 PF->KNI Rx queue->KNI raw socket queue->Guest VM virtio port 0 Rx burst->Guest VM virtio port 0 Tx burst-> KNI Tx queue ->82599 PF-> IXIA packet generator

9.51.5 Virtio with qemu virtio Back End

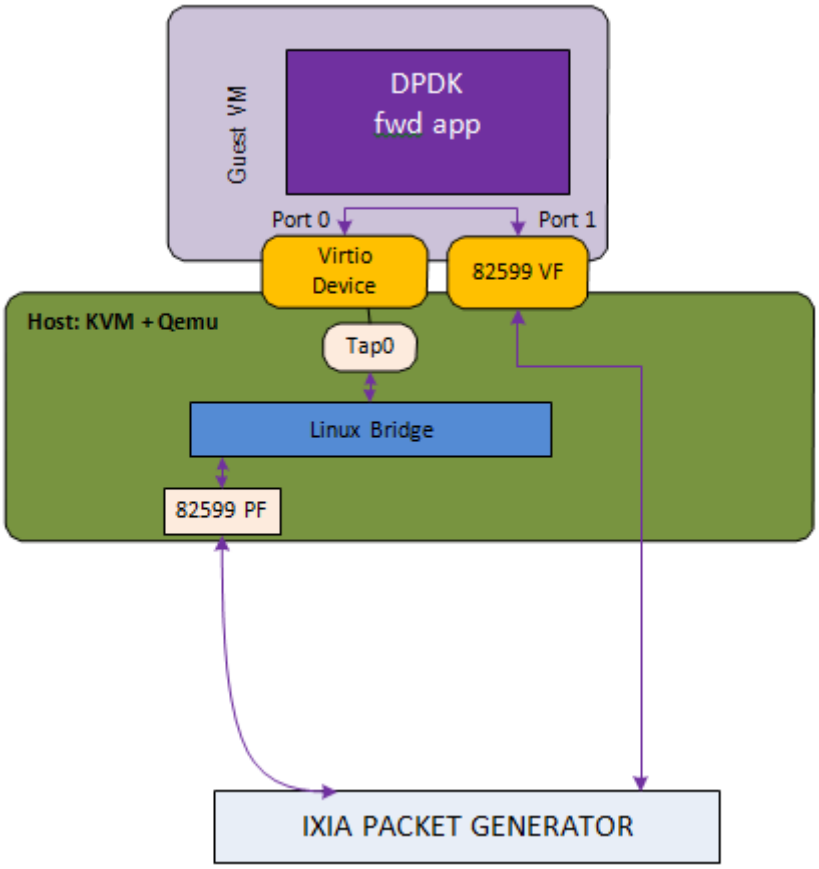


Fig. 9.11: Host2VM Communication Example Using qemu vhost Back End

```
qemu-system-x86_64 -enable-kvm -cpu host -m 2048 -smp 2 -mem-path /dev/
hugepages -mem-prealloc
-drive file=/data/DPDKVMS/dpdk-vm1
-netdev tap,id=vm1_p1,ifname=tap0,script=no,vhost=on
-device virtio-net-pci,netdev=vm1_p1,bus=pci.0,addr=0x3,ioeventfd=on
-device pci-assign,host=04:10.1 \
```

In this example, the packet reception flow path is:

> IXIA packet generator->82599 PF->Linux Bridge->TAP0's socket queue-> Guest VM virtio port 0 Rx burst-> Guest VM 82599 VF port1 Tx burst-> IXIA packet generator

The packet transmission flow is:

> IXIA packet generator-> Guest VM 82599 VF port1 Rx burst-> Guest VM virtio port 0 Tx burst-> tap -> Linux Bridge->82599 PF-> IXIA packet generator

9.51.6 Virtio PMD Rx/Tx Callbacks

Virtio driver has 6 Rx callbacks and 3 Tx callbacks.

Rx callbacks:

1. `virtio_recv_pkts`: Regular version without mergeable Rx buffer support for split virtqueue.
2. `virtio_recv_mergeable_pkts`: Regular version with mergeable Rx buffer support for split virtqueue.
3. `virtio_recv_pkts_vec`: Vector version without mergeable Rx buffer support, also fixes the available ring indexes and uses vector instructions to optimize performance for split virtqueue.
4. `virtio_recv_pkts_inorder`: In-order version with mergeable and non-mergeable Rx buffer support for split virtqueue.
5. `virtio_recv_pkts_packed`: Regular and in-order version without mergeable Rx buffer support for packed virtqueue.
6. `virtio_recv_mergeable_pkts_packed`: Regular and in-order version with mergeable Rx buffer support for packed virtqueue.

Tx callbacks:

1. `virtio_xmit_pkts`: Regular version for split virtqueue.
2. `virtio_xmit_pkts_inorder`: In-order version for split virtqueue.
3. `virtio_xmit_pkts_packed`: Regular and in-order version for packed virtqueue.

By default, the non-vector callbacks are used:

- For Rx: If mergeable Rx buffers is disabled then `virtio_recv_pkts` or `virtio_recv_pkts_packed` will be used, otherwise `virtio_recv_mergeable_pkts` or `virtio_recv_mergeable_pkts_packed` will be used.
- For Tx: `virtio_xmit_pkts` or `virtio_xmit_pkts_packed` will be used.

Vector callbacks will be used when:

- Mergeable Rx buffers is disabled.

The corresponding callbacks are:

- For Rx: `virtio_recv_pkts_vec`.

There is no vector callbacks for packed virtqueue for now.

Example of using the vector version of the virtio poll mode driver in `testpmd`:

```
testpmd -l 0-2 -n 4 -- -i --rxq=1 --txq=1 --nb-cores=1
```

In-order callbacks only work on simulated virtio user vdev.

For split virtqueue:

- For Rx: If in-order is enabled then `virtio_recv_pkts_inorder` is used.
- For Tx: If in-order is enabled then `virtio_xmit_pkts_inorder` is used.

For packed virtqueue, the default callbacks already support the in-order feature.

9.51.7 Interrupt mode

There are three kinds of interrupts from a virtio device over PCI bus: config interrupt, Rx interrupts, and Tx interrupts. Config interrupt is used for notification of device configuration changes, especially link status (lsc). Interrupt mode is translated into Rx interrupts in the context of DPDK.

Note: Virtio PMD already has support for receiving lsc from qemu when the link status changes, especially when vhost user disconnects. However, it fails to do that if the VM is created by qemu 2.6.2 or below, since the capability to detect vhost user disconnection is introduced in qemu 2.7.0.

Prerequisites for Rx interrupts

To support Rx interrupts, #. Check if guest kernel supports VFIO-NOIOMMU:

Linux started to support VFIO-NOIOMMU since 4.8.0. Make sure the guest kernel is compiled with:

```
CONFIG_VFIO_NOIOMMU=y
```

1. Properly set msix vectors when starting VM:

   Enable multi-queue when starting VM, and specify msix vectors in qemu cmdline. (N+1) is the minimum, and (2N+2) is mostly recommended.

   ```
   $(QEMU) ... -device virtio-net-pci,mq=on,vectors=2N+2 ...
   ```

2. In VM, insert vfio module in NOIOMMU mode:

   ```
   modprobe vfio enable_unsafe_noiommu_mode=1
   modprobe vfio-pci
   ```

3. In VM, bind the virtio device with vfio-pci:

   ```
   python usertools/dpdk-devbind.py -b vfio-pci 00:03.0
   ```

Example

Here we use l3fwd-power as an example to show how to get started.

Example:

```
$ l3fwd-power -l 0-1 -- -p 1 -P --config="(0,0,1)" \
                                    --no-numa --parse-ptype
```

9.51.8 Virtio PMD arguments

Below devargs are supported by the PCI virtio driver:

1. `vdpa`:

   A virtio device could also be driven by vDPA (vhost data path acceleration) driver, and works as a HW vhost backend. This argument is used to specify a virtio device needs to work in vDPA mode. (Default: 0 (disabled))

2. `speed`:

   It is used to specify link speed of virtio device. Link speed is a part of link status structure. It could be requested by application using rte_eth_link_get_nowait function. (Default: 10000 (10G))

3. `vectorized`:

   It is used to specify whether virtio device prefers to use vectorized path. Afterwards, dependencies of vectorized path will be checked in path election. (Default: 0 (disabled))

Below devargs are supported by the virtio-user vdev:

1. `path`:

   It is used to specify a path to connect to vhost backend.

2. `mac`:

   It is used to specify the MAC address.

3. `cq`:

   It is used to enable the control queue. (Default: 0 (disabled))

4. `queue_size`:

   It is used to specify the queue size. (Default: 256)

5. `queues`:

   It is used to specify the queue number. (Default: 1)

6. `iface`:

   It is used to specify the host interface name for vhost-kernel backend.

7. `server`:

   It is used to enable the server mode when using vhost-user backend. (Default: 0 (disabled))

8. `mrg_rxbuf`:

   It is used to enable virtio device mergeable Rx buffer feature. (Default: 1 (enabled))

9. `in_order`:

   It is used to enable virtio device in-order feature. (Default: 1 (enabled))

10. `packed_vq`:

    It is used to enable virtio device packed virtqueue feature. (Default: 0 (disabled))

11. `speed`:

    It is used to specify link speed of virtio device. Link speed is a part of link status structure. It could be requested by application using rte_eth_link_get_nowait function. (Default: 10000 (10G))

12. `vectorized`:

    It is used to specify whether virtio device prefers to use vectorized path. Afterwards, dependencies of vectorized path will be checked in path election. (Default: 0 (disabled))

9.51.9 Virtio paths Selection and Usage

Logically virtio-PMD has 9 paths based on the combination of virtio features (Rx mergeable, In-order, Packed virtqueue), below is an introduction of these features:

- Rx mergeable: With this feature negotiated, device can receive large packets by combining individual descriptors.
- In-order: Some devices always use descriptors in the same order in which they have been made available, these devices can offer the VIRTIO_F_IN_ORDER feature. With this feature negotiated, driver will use descriptors in order.
- Packed virtqueue: The structure of packed virtqueue is different from split virtqueue, split virtqueue is composed of available ring, used ring and descriptor table, while packed virtqueue is composed of descriptor ring, driver event suppression and device event suppression. The idea behind this is to improve performance by avoiding cache misses and make it easier for hardware to implement.

Virtio paths Selection

If packed virtqueue is not negotiated, below split virtqueue paths will be selected according to below configuration:

1. Split virtqueue mergeable path: If Rx mergeable is negotiated, in-order feature is not negotiated, this path will be selected.
2. Split virtqueue non-mergeable path: If Rx mergeable and in-order feature are not negotiated, also Rx offload(s) are requested, this path will be selected.
3. Split virtqueue in-order mergeable path: If Rx mergeable and in-order feature are both negotiated, this path will be selected.
4. Split virtqueue in-order non-mergeable path: If in-order feature is negotiated and Rx mergeable is not negotiated, this path will be selected.
5. Split virtqueue vectorized Rx path: If Rx mergeable is disabled and no Rx offload requested, this path will be selected.

If packed virtqueue is negotiated, below packed virtqueue paths will be selected according to below configuration:

1. Packed virtqueue mergeable path: If Rx mergeable is negotiated, in-order feature is not negotiated, this path will be selected.
2. Packed virtqueue non-mergeable path: If Rx mergeable and in-order feature are not negotiated, this path will be selected.
3. Packed virtqueue in-order mergeable path: If in-order and Rx mergeable feature are both negotiated, this path will be selected.
4. Packed virtqueue in-order non-mergeable path: If in-order feature is negotiated and Rx mergeable is not negotiated, this path will be selected.

5. Packed virtqueue vectorized Rx path: If building and running environment support AVX512 && in-order feature is negotiated && Rx mergeable is not negotiated && TCP_LRO Rx offloading is disabled && vectorized option enabled, this path will be selected.
6. Packed virtqueue vectorized Tx path: If building and running environment support AVX512 && in-order feature is negotiated && vectorized option enabled, this path will be selected.

Rx/Tx callbacks of each Virtio path

Refer to above description, virtio path and corresponding Rx/Tx callbacks will be selected automatically. Rx callbacks and Tx callbacks for each virtio path are shown in below table:

Table 9.18: Virtio Paths and Callbacks

Virtio paths	Rx callbacks	Tx callbacks
Split virtqueue mergeable path	virtio_recv_mergeable_pkts	virtio_xmit_pkts
Split virtqueue non-mergeable path	virtio_recv_pkts	virtio_xmit_pkts
Split virtqueue in-order mergeable path	virtio_recv_pkts_inorder	virtio_xmit_pkts_inorder
Split virtqueue in-order non-mergeable path	virtio_recv_pkts_inorder	virtio_xmit_pkts_inorder
Split virtqueue vectorized Rx path	virtio_recv_pkts_vec	virtio_xmit_pkts
Packed virtqueue mergeable path	virtio_recv_mergeable_pkts_packed	virtio_xmit_pkts_packed
Packed virtqueue non-meregable path	virtio_recv_pkts_packed	virtio_xmit_pkts_packed
Packed virtqueue in-order mergeable path	virtio_recv_mergeable_pkts_packed	virtio_xmit_pkts_packed
Packed virtqueue in-order non-mergeable path	virtio_recv_pkts_packed	virtio_xmit_pkts_packed
Packed virtqueue vectorized Rx path	virtio_recv_pkts_packed_vec	virtio_xmit_pkts_packed
Packed virtqueue vectorized Tx path	virtio_recv_pkts_packed	virtio_xmit_pkts_packed_vec

Virtio paths Support Status from Release to Release

Virtio feature implementation:

- In-order feature is supported since DPDK 18.08 by adding new Rx/Tx callbacks `virtio_recv_pkts_inorder` and `virtio_xmit_pkts_inorder`.
- Packed virtqueue is supported since DPDK 19.02 by adding new Rx/Tx callbacks `virtio_recv_pkts_packed` , `virtio_recv_mergeable_pkts_packed` and `virtio_xmit_pkts_packed`.

All virtio paths support status are shown in below table:

Table 9.19: Virtio Paths and Releases

Virtio paths	16.11 ~ 18.05	18.08 ~ 18.11	19.02 ~ 19.11	20.05 ~
Split virtqueue mergeable path	Y	Y	Y	Y
Split virtqueue non-mergeable path	Y	Y	Y	Y
Split virtqueue vectorized Rx path	Y	Y	Y	Y
Split virtqueue simple Tx path	Y	N	N	N
Split virtqueue in-order mergeable path		Y	Y	Y
Split virtqueue in-order non-mergeable path		Y	Y	Y
Packed virtqueue mergeable path			Y	Y
Packed virtqueue non-mergeable path			Y	Y
Packed virtqueue in-order mergeable path			Y	Y
Packed virtqueue in-order non-mergeable path			Y	Y
Packed virtqueue vectorized Rx path				Y
Packed virtqueue vectorized Tx path				Y

QEMU Support Status

- Qemu now supports three paths of split virtqueue: Split virtqueue mergeable path, Split virtqueue non-mergeable path, Split virtqueue vectorized Rx path.
- Since qemu 4.2.0, Packed virtqueue mergeable path and Packed virtqueue non-mergeable path can be supported.

How to Debug

If you meet performance drop or some other issues after upgrading the driver or configuration, below steps can help you identify which path you selected and root cause faster.

1. Run vhost/virtio test case;
2. Run “perf top” and check virtio Rx/Tx callback names;
3. Identify which virtio path is selected refer to above table.

9.52 Poll Mode Driver that wraps vhost library

This PMD is a thin wrapper of the DPDK vhost library. The user can handle virtqueues as one of normal DPDK port.

9.52.1 Vhost Implementation in DPDK

Please refer to Chapter "Vhost Library" of *DPDK Programmer's Guide* to know detail of vhost.

9.52.2 Features and Limitations of vhost PMD

Currently, the vhost PMD provides the basic functionality of packet reception, transmission and event handling.

- It has multiple queues support.
- It supports `RTE_ETH_EVENT_INTR_LSC` and `RTE_ETH_EVENT_QUEUE_STATE` events.
- It supports Port Hotplug functionality.
- Don't need to stop RX/TX, when the user wants to stop a guest or a virtio-net driver on guest.

9.52.3 Vhost PMD arguments

The user can specify below arguments in *–vdev* option.

1. `iface`:

   It is used to specify a path to connect to a QEMU virtio-net device.

2. `queues`:

   It is used to specify the number of queues virtio-net device has. (Default: 1)

3. `iommu-support`:

   It is used to enable iommu support in vhost library. (Default: 0 (disabled))

4. `postcopy-support`:

   It is used to enable postcopy live-migration support in vhost library. (Default: 0 (disabled))

5. `tso`:

   It is used to enable tso support in vhost library. (Default: 0 (disabled))

6. `linear-buffer`:

   It is used to enable linear buffer support in vhost library. (Default: 0 (disabled))

7. `ext-buffer`:

   It is used to enable external buffer support in vhost library. (Default: 0 (disabled))

9.52.4 Vhost PMD event handling

This section describes how to handle vhost PMD events.

The user can register an event callback handler with `rte_eth_dev_callback_register()`. The registered callback handler will be invoked with one of below event types.

1. `RTE_ETH_EVENT_INTR_LSC`:

   It means link status of the port was changed.

2. `RTE_ETH_EVENT_QUEUE_STATE`:

   It means some of queue statuses were changed. Call `rte_eth_vhost_get_queue_event()` in the callback handler. Because changing multiple statuses may occur only one event, call the function repeatedly as long as it doesn't return negative value.

9.52.5 Vhost PMD with testpmd application

This section demonstrates vhost PMD with testpmd DPDK sample application.

1. Launch the testpmd with vhost PMD:

   ```
   ./testpmd -l 0-3 -n 4 --vdev 'net_vhost0,iface=/tmp/sock0,queues=1' -- -i
   ```

   Other basic DPDK preparations like hugepage enabling here. Please refer to the *DPDK Getting Started Guide* for detailed instructions.

2. Launch the QEMU:

   ```
   qemu-system-x86_64 <snip>
               -chardev socket,id=chr0,path=/tmp/sock0 \
               -netdev vhost-user,id=net0,chardev=chr0,vhostforce,queues=1 \
               -device virtio-net-pci,netdev=net0
   ```

   This command attaches one virtio-net device to QEMU guest. After initialization processes between QEMU and DPDK vhost library are done, status of the port will be linked up.

9.53 Poll Mode Driver for Paravirtual VMXNET3 NIC

The VMXNET3 adapter is the next generation of a paravirtualized NIC, introduced by VMware* ESXi. It is designed for performance, offers all the features available in VMXNET2, and adds several new features such as, multi-queue support (also known as Receive Side Scaling, RSS), IPv6 offloads, and MSI/MSI-X interrupt delivery. One can use the same device in a DPDK application with VMXNET3 PMD introduced in DPDK API.

In this chapter, two setups with the use of the VMXNET3 PMD are demonstrated:

1. Vmxnet3 with a native NIC connected to a vSwitch
2. Vmxnet3 chaining VMs connected to a vSwitch

9.53.1 VMXNET3 Implementation in the DPDK

For details on the VMXNET3 device, refer to the VMXNET3 driver's vmxnet3 directory and support manual from VMware*.

For performance details, refer to the following link from VMware:

http://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf

As a PMD, the VMXNET3 driver provides the packet reception and transmission callbacks, vmxnet3_recv_pkts and vmxnet3_xmit_pkts.

The VMXNET3 PMD handles all the packet buffer memory allocation and resides in guest address space and it is solely responsible to free that memory when not needed. The packet buffers and features to be supported are made available to hypervisor via VMXNET3 PCI configuration space BARs. During RX/TX, the packet buffers are exchanged by their GPAs, and the hypervisor loads the buffers with packets in the RX case and sends packets to vSwitch in the TX case.

The VMXNET3 PMD is compiled with vmxnet3 device headers. The interface is similar to that of the other PMDs available in the DPDK API. The driver pre-allocates the packet buffers and loads the command ring descriptors in advance. The hypervisor fills those packet buffers on packet arrival and write completion ring descriptors, which are eventually pulled by the PMD. After reception, the DPDK application frees the descriptors and loads new packet buffers for the coming packets. The interrupts are disabled and there is no notification required. This keeps performance up on the RX side, even though the device provides a notification feature.

In the transmit routine, the DPDK application fills packet buffer pointers in the descriptors of the command ring and notifies the hypervisor. In response the hypervisor takes packets and passes them to the vSwitch, It writes into the completion descriptors ring. The rings are read by the PMD in the next transmit routine call and the buffers and descriptors are freed from memory.

9.53.2 Features and Limitations of VMXNET3 PMD

In release 1.6.0, the VMXNET3 PMD provides the basic functionality of packet reception and transmission. There are several options available for filtering packets at VMXNET3 device level including:

1. MAC Address based filtering:
   - Unicast, Broadcast, All Multicast modes - SUPPORTED BY DEFAULT
   - Multicast with Multicast Filter table - NOT SUPPORTED
   - Promiscuous mode - SUPPORTED
   - RSS based load balancing between queues - SUPPORTED
2. VLAN filtering:
   - VLAN tag based filtering without load balancing - SUPPORTED

Note:

- Release 1.6.0 does not support separate headers and body receive cmd_ring and hence, multiple segment buffers are not supported. Only cmd_ring_0 is used for packet buffers, one for each descriptor.
- Receive and transmit of scattered packets is not supported.

- Multicast with Multicast Filter table is not supported.

9.53.3 Prerequisites

The following prerequisites apply:

- Before starting a VM, a VMXNET3 interface to a VM through VMware vSphere Client must be assigned. This is shown in the figure below.

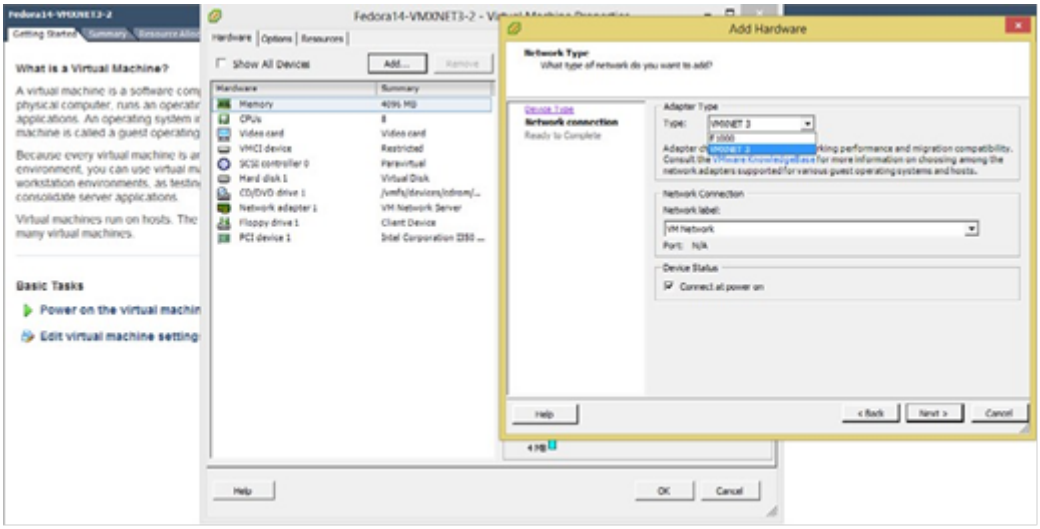


Fig. 9.12: Assigning a VMXNET3 interface to a VM using VMware vSphere Client

Note: Depending on the Virtual Machine type, the VMware vSphere Client shows Ethernet adaptors while adding an Ethernet device. Ensure that the VM type used offers a VMXNET3 device. Refer to the VMware documentation for a listed of VMs.

Note: Follow the *DPDK Getting Started Guide* to setup the basic DPDK environment.

Note: Follow the *DPDK Sample Application's User Guide*, L2 Forwarding/L3 Forwarding and TestPMD for instructions on how to run a DPDK application using an assigned VMXNET3 device.

9.53.4 VMXNET3 with a Native NIC Connected to a vSwitch

This section describes an example setup for Phy-vSwitch-VM-Phy communication.

Note: Other instructions on preparing to use DPDK such as, hugepage enabling, uio port binding are not listed here. Please refer to *DPDK Getting Started Guide and DPDK Sample Application's User Guide* for detailed instructions.

The packet reception and transmission flow path is:

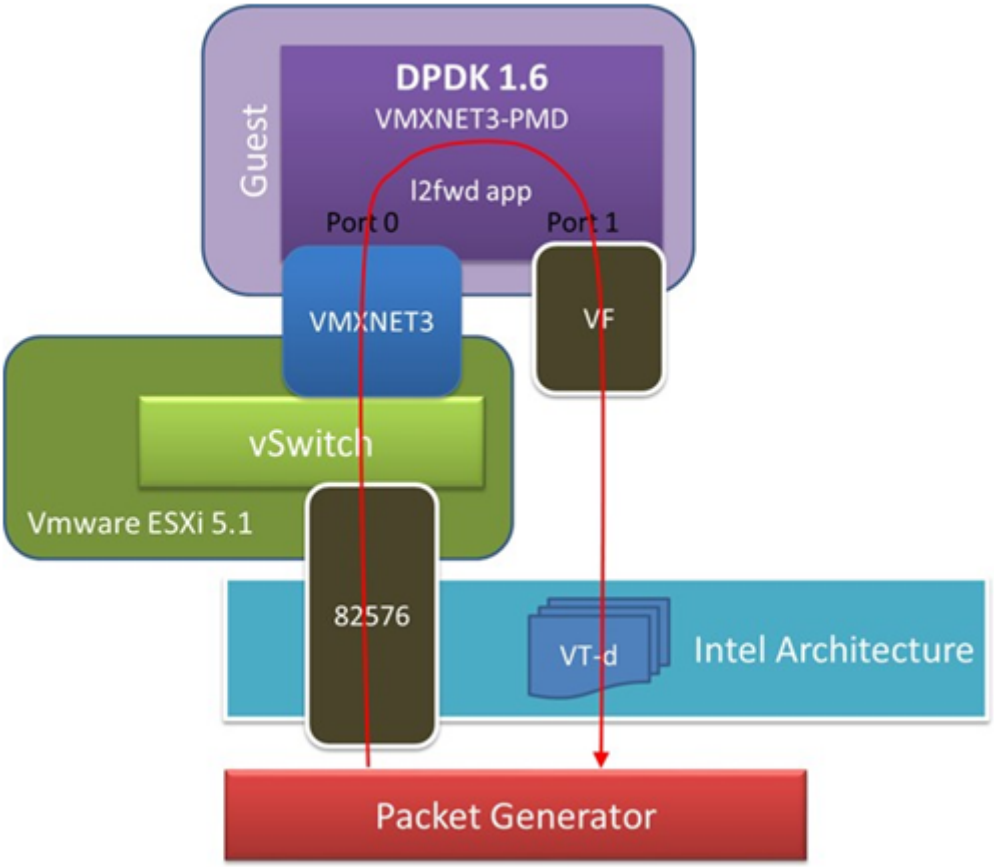


Fig. 9.13: VMXNET3 with a Native NIC Connected to a vSwitch

```
Packet generator -> 82576
                 -> VMware ESXi vSwitch
                 -> VMXNET3 device
                 -> Guest VM VMXNET3 port 0 rx burst
                 -> Guest VM 82599 VF port 0 tx burst
                 -> 82599 VF
                 -> Packet generator
```

9.53.5 VMXNET3 Chaining VMs Connected to a vSwitch

The following figure shows an example VM-to-VM communication over a Phy-VM-vSwitch-VM-Phy communication channel.

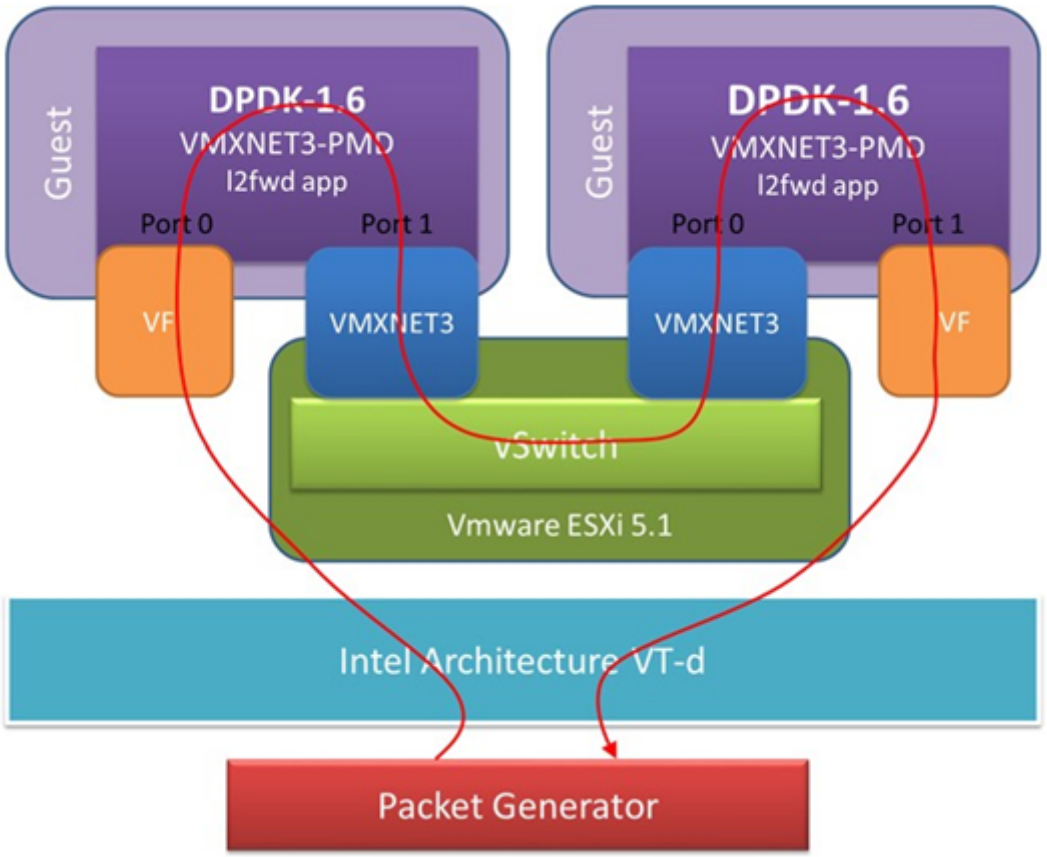


Fig. 9.14: VMXNET3 Chaining VMs Connected to a vSwitch

Note: When using the L2 Forwarding or L3 Forwarding applications, a destination MAC address needs to be written in packets to hit the other VM's VMXNET3 interface.

In this example, the packet flow path is:

```
Packet generator -> 82599 VF
                 -> Guest VM 82599 port 0 rx burst
                 -> Guest VM VMXNET3 port 1 tx burst
                 -> VMXNET3 device
                 -> VMware ESXi vSwitch
```

(continues on next page)

(continued from previous page)

```
                -> VMXNET3 device
                -> Guest VM VMXNET3 port 0 rx burst
                -> Guest VM 82599 VF port 1 tx burst
                -> 82599 VF
                -> Packet generator
```

9.54 Libpcap and Ring Based Poll Mode Drivers

In addition to Poll Mode Drivers (PMDs) for physical and virtual hardware, the DPDK also includes pure-software PMDs, two of these drivers are:

- A libpcap -based PMD (librte_pmd_pcap) that reads and writes packets using libpcap, - both from files on disk, as well as from physical NIC devices using standard Linux kernel drivers.
- A ring-based PMD (librte_pmd_ring) that allows a set of software FIFOs (that is, rte_ring) to be accessed using the PMD APIs, as though they were physical NICs.

Note: The libpcap -based PMD is disabled by default in the build configuration files, owing to an external dependency on the libpcap development files which must be installed on the board. Once the libpcap development files are installed, the library can be enabled by setting CONFIG_RTE_LIBRTE_PMD_PCAP=y and recompiling the DPDK.

9.54.1 Using the Drivers from the EAL Command Line

For ease of use, the DPDK EAL also has been extended to allow pseudo-Ethernet devices, using one or more of these drivers, to be created at application startup time during EAL initialization.

To do so, the –vdev= parameter must be passed to the EAL. This takes take options to allow ring and pcap-based Ethernet to be allocated and used transparently by the application. This can be used, for example, for testing on a virtual machine where there are no Ethernet ports.

Libpcap-based PMD

Pcap-based devices can be created using the virtual device –vdev option. The device name must start with the net_pcap prefix followed by numbers or letters. The name is unique for each device. Each device can have multiple stream options and multiple devices can be used. Multiple device definitions can be arranged using multiple –vdev. Device name and stream options must be separated by commas as shown below:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,stream_opt0=..,stream_opt1=..' \
    --vdev='net_pcap1,stream_opt0=..'
```

Device Streams

Multiple ways of stream definitions can be assessed and combined as long as the following two rules are respected:

- A device is provided with two different streams - reception and transmission.
- A device is provided with one network interface name used for reading and writing packets.

The different stream types are:

- rx_pcap: Defines a reception stream based on a pcap file. The driver reads each packet within the given pcap file as if it was receiving it from the wire. The value is a path to a valid pcap file.

  rx_pcap=/path/to/file.pcap

- tx_pcap: Defines a transmission stream based on a pcap file. The driver writes each received packet to the given pcap file. The value is a path to a pcap file. The file is overwritten if it already exists and it is created if it does not.

  tx_pcap=/path/to/file.pcap

- rx_iface: Defines a reception stream based on a network interface name. The driver reads packets from the given interface using the Linux kernel driver for that interface. The driver captures both the incoming and outgoing packets on that interface. The value is an interface name.

  rx_iface=eth0

- rx_iface_in: Defines a reception stream based on a network interface name. The driver reads packets from the given interface using the Linux kernel driver for that interface. The driver captures only the incoming packets on that interface. The value is an interface name.

  rx_iface_in=eth0

- tx_iface: Defines a transmission stream based on a network interface name. The driver sends packets to the given interface using the Linux kernel driver for that interface. The value is an interface name.

  tx_iface=eth0

- iface: Defines a device mapping a network interface. The driver both reads and writes packets from and to the given interface. The value is an interface name.

  iface=eth0

Runtime Config Options

- Use PCAP interface physical MAC

  In case `iface=` configuration is set, user may want to use the selected interface's physical MAC address. This can be done with a `devarg` `phy_mac`, for example:

```
--vdev 'net_pcap0,iface=eth0,phy_mac=1'
```

- Use the RX PCAP file to infinitely receive packets

  In case `rx_pcap=` configuration is set, user may want to use the selected PCAP file for rudimental performance testing. This can be done with a `devarg` `infinite_rx`, for example:

```
--vdev 'net_pcap0,rx_pcap=file_rx.pcap,infinite_rx=1'
```

When this mode is used, it is recommended to drop all packets on transmit by not providing a tx_pcap or tx_iface.

This option is device wide, so all queues on a device will either have this enabled or disabled. This option should only be provided once per device.

- Drop all packets on transmit

  The user may want to drop all packets on tx for a device. This can be done by not providing a tx_pcap or tx_iface, for example:

```
--vdev 'net_pcap0,rx_pcap=file_rx.pcap'
```

  In this case, one tx drop queue is created for each rxq on that device.

  - Receive no packets on Rx

  The user may want to run without receiving any packets on Rx. This can be done by not providing a rx_pcap or rx_iface, for example:

```
--vdev 'net_pcap0,tx_pcap=file_tx.pcap'
```

In this case, one dummy rx queue is created for each tx queue argument passed

Examples of Usage

Read packets from one pcap file and write them to another:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,rx_pcap=file_rx.pcap,tx_pcap=file_tx.pcap' \
    -- --port-topology=chained
```

Read packets from a network interface and write them to a pcap file:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,rx_iface=eth0,tx_pcap=file_tx.pcap' \
    -- --port-topology=chained
```

Read packets from a pcap file and write them to a network interface:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,rx_pcap=file_rx.pcap,tx_iface=eth1' \
    -- --port-topology=chained
```

Forward packets through two network interfaces:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,iface=eth0' --vdev='net_pcap1;iface=eth1'
```

Enable 2 tx queues on a network interface:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,rx_iface=eth1,tx_iface=eth1,tx_iface=eth1' \
    -- --txq 2
```

Read only incoming packets from a network interface and write them back to the same network interface:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,rx_iface_in=eth1,tx_iface=eth1'
```

Using libpcap-based PMD with the testpmd Application

One of the first things that testpmd does before starting to forward packets is to flush the RX streams by reading the first 512 packets on every RX stream and discarding them. When using a libpcap-based PMD this behavior can be turned off using the following command line option:

```
--no-flush-rx
```

It is also available in the runtime command line:

```
set flush_rx on/off
```

It is useful for the case where the rx_pcap is being used and no packets are meant to be discarded. Otherwise, the first 512 packets from the input pcap file will be discarded by the RX flushing operation.

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
    --vdev 'net_pcap0,rx_pcap=file_rx.pcap,tx_pcap=file_tx.pcap' \
    -- --port-topology=chained --no-flush-rx
```

Note: The network interface provided to the PMD should be up. The PMD will return an error if interface is down, and the PMD itself won't change the status of the external network interface.

Rings-based PMD

To run a DPDK application on a machine without any Ethernet devices, a pair of ring-based rte_ethdevs can be used as below. The device names passed to the –vdev option must start with net_ring and take no additional parameters. Multiple devices may be specified, separated by commas.

```
./testpmd -l 1-3 -n 4 --vdev=net_ring0 --vdev=net_ring1 -- -i
EAL: Detected lcore 1 as core 1 on socket 0
...

Interactive-mode selected
Configuring Port 0 (socket 0)
Configuring Port 1 (socket 0)
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done

testpmd> start tx_first
io packet forwarding - CRC stripping disabled - packets/burst=16
nb forwarding cores=1 - nb forwarding ports=2
RX queues=1 - RX desc=128 - RX free threshold=0
RX threshold registers: pthresh=8 hthresh=8 wthresh=4
TX queues=1 - TX desc=512 - TX free threshold=0
TX threshold registers: pthresh=36 hthresh=0 wthresh=0
TX RS bit threshold=0 - TXQ flags=0x0
```

(continues on next page)

(continued from previous page)

```
testpmd> stop
Telling cores to stop...
Waiting for lcores to finish...
```

```
---------------------- Forward statistics for port 0  ----------------------
RX-packets: 231192368      RX-dropped: 0             RX-total: 231192368
TX-packets: 231192384      TX-dropped: 0             TX-total: 231192384
----------------------------------------------------------------------------

---------------------- Forward statistics for port 1  ----------------------
RX-packets: 231192368      RX-dropped: 0             RX-total: 231192368
TX-packets: 231192384      TX-dropped: 0             TX-total: 231192384
----------------------------------------------------------------------------
```

```
++++++++++++++ Accumulated forward statistics for allports++++++++++
RX-packets: 462384736  RX-dropped: 0 RX-total: 462384736
TX-packets: 462384768  TX-dropped: 0 TX-total: 462384768
++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Done.
```

Using the Poll Mode Driver from an Application

Both drivers can provide similar APIs to allow the user to create a PMD, that is, rte_ethdev structure, instances at run-time in the end-application, for example, using rte_eth_from_rings() or rte_eth_from_pcaps() APIs. For the rings-based PMD, this functionality could be used, for example, to allow data exchange between cores using rings to be done in exactly the same way as sending or receiving packets from an Ethernet device. For the libpcap-based PMD, it allows an application to open one or more pcap files and use these as a source of packet input to the application.

Usage Examples

To create two pseudo-Ethernet ports where all traffic sent to a port is looped back for reception on the same port (error handling omitted for clarity):

```
#define RING_SIZE 256
#define NUM_RINGS 2
#define SOCKET0 0

struct rte_ring *ring[NUM_RINGS];
int port0, port1;

ring[0] = rte_ring_create("R0", RING_SIZE, SOCKET0, RING_F_SP_ENQ|RING_F_SC_DEQ);
ring[1] = rte_ring_create("R1", RING_SIZE, SOCKET0, RING_F_SP_ENQ|RING_F_SC_DEQ);

/* create two ethdev's */

port0 = rte_eth_from_rings("net_ring0", ring, NUM_RINGS, ring, NUM_RINGS, SOCKET0);
port1 = rte_eth_from_rings("net_ring1", ring, NUM_RINGS, ring, NUM_RINGS, SOCKET0);
```

To create two pseudo-Ethernet ports where the traffic is switched between them, that is, traffic sent to port 0 is read back from port 1 and vice-versa, the final two lines could be changed as below:

```
port0 = rte_eth_from_rings("net_ring0", &ring[0], 1, &ring[1], 1, SOCKET0);
port1 = rte_eth_from_rings("net_ring1", &ring[1], 1, &ring[0], 1, SOCKET0);
```

This type of configuration could be useful in a pipeline model, for example, where one may want to have inter-core communication using pseudo Ethernet devices rather than raw rings, for reasons of API consistency.

Enqueuing and dequeuing items from an rte_ring using the rings-based PMD may be slower than using the native rings API. This is because DPDK Ethernet drivers make use of function pointers to call the appropriate enqueue or dequeue functions, while the rte_ring specific functions are direct function calls in the code and are often inlined by the compiler.

> Once an ethdev has been created, for either a ring or a pcap-based PMD, it should be configured and started in the same way as a regular Ethernet device, that is, by calling rte_eth_dev_configure() to set the number of receive and transmit queues, then calling rte_eth_rx_queue_setup() / tx_queue_setup() for each of those queues and finally calling rte_eth_dev_start() to allow transmission and reception of packets to begin.

9.55 Fail-safe poll mode driver library

The Fail-safe poll mode driver library (**librte_pmd_failsafe**) implements a virtual device that allows using device supporting hotplug, without modifying other components relying on such device (application, other PMDs). In this context, hotplug support is meant as plugging or removing a device from its bus suddenly.

Additionally to the Seamless Hotplug feature, the Fail-safe PMD offers the ability to redirect operations to a secondary device when the primary has been removed from the system.

Note: The library is enabled by default. You can enable it or disable it manually by setting the `CONFIG_RTE_LIBRTE_PMD_FAILSAFE` configuration option.

9.55.1 Features

The Fail-safe PMD only supports a limited set of features. If you plan to use a device underneath the Fail-safe PMD with a specific feature, this feature must also be supported by the Fail-safe PMD.

A notable exception is the device removal feature. The fail-safe PMD is not meant to be removed itself, unlike its sub-devices which should support it. If a sub-device supports hotplugging, the fail-safe PMD will enable its use automatically by detecting capable devices and registering the relevant handler.

Check the feature matrix for the complete set of supported features.

9.55.2 Compilation option

Available options within the `$RTE_TARGET/build/.config` file:

- `CONFIG_RTE_LIBRTE_PMD_FAILSAFE` (default **y**)

  This option enables or disables compiling librte_pmd_failsafe.

9.55.3 Using the Fail-safe PMD from the EAL command line

The Fail-safe PMD can be used like most other DPDK virtual devices, by passing a `--vdev` parameter to the EAL when starting the application. The device name must start with the *net_failsafe* prefix, followed by numbers or letters. This name must be unique for each device. Each fail-safe instance must have at least one sub-device, and at most two.

A sub-device can be any DPDK device, including possibly another fail-safe device.

Fail-safe command line parameters

- **dev(<iface>)** parameter

  This parameter allows the user to define a sub-device. The `<iface>` part of this parameter must be a valid device definition. It follows the same format provided to any `-w` or `--vdev` options.

  Enclosing the device definition within parentheses here allows using additional sub-device parameters if need be. They will be passed on to the sub-device.

Note: In case where the sub-device is also used as a whitelist device, using `-w` on the EAL command line, the fail-safe PMD will use the device with the options provided to the EAL instead of its own parameters.

When trying to use a PCI device automatically probed by the blacklist mode, the name for the fail-safe sub-device must be the full PCI id: Domain:Bus:Device.Function, *i.e.* `00:00:00.0` instead of `00:00.0`, as the second form is historically accepted by the DPDK.

- **exec(<shell command>)** parameter

  This parameter allows the user to provide a command to the fail-safe PMD to execute and define a sub-device. It is done within a regular shell context. The first line of its output is read by the fail-safe PMD and otherwise interpreted as if passed to a **dev** parameter. Any other line is discarded. If the command fails or output an incorrect string, the sub-device is not initialized. All commas within the `shell command` are replaced by spaces before executing the command. This helps using scripts to specify devices.

- **fd(<file descriptor number>)** parameter

  This parameter reads a device definition from an arbitrary file descriptor number in `<iface>` format as described above.

  The file descriptor is read in non-blocking mode and is never closed in order to take only the last line into account (unlike `exec()`) at every probe attempt.

- **mac** parameter [MAC address]

  This parameter allows the user to set a default MAC address to the fail-safe and all of its sub-devices. If no default mac address is provided, the fail-safe PMD will read the MAC address of the

first of its sub-device to be successfully probed and use it as its default MAC address, trying to set it to all of its other sub-devices. If no sub-device was successfully probed at initialization, then a random MAC address is generated, that will be subsequently applied to all sub-devices once they are probed.

- **hotplug_poll** parameter [UINT64] (default **2000**)

  This parameter allows the user to configure the amount of time in milliseconds between two sub-device upkeep round.

Usage example

This section shows some example of using **testpmd** with a fail-safe PMD.

1. To build a PMD and configure DPDK, refer to the document *compiling and testing a PMD for a NIC*.

2. Start testpmd. The sub-device `84:00.0` should be blacklisted from normal EAL operations to avoid probing it twice, as the PCI bus is in blacklist mode.

```
$RTE_TARGET/build/app/testpmd -c 0xff -n 4 \
   --vdev 'net_failsafe0,mac=de:ad:be:ef:01:02,dev(84:00.0),dev(net_ring0)' \
   -b 84:00.0 -b 00:04.0 -- -i
```

   If the sub-device `84:00.0` is not blacklisted, it will be probed by the EAL first. When the fail-safe then tries to initialize it the probe operation fails.

   Note that PCI blacklist mode is the default PCI operating mode.

3. Alternatively, it can be used alongside any other device in whitelist mode.

```
$RTE_TARGET/build/app/testpmd -c 0xff -n 4 \
   --vdev 'net_failsafe0,mac=de:ad:be:ef:01:02,dev(84:00.0),dev(net_ring0)' \
   -w 81:00.0 -- -i
```

4. Start testpmd using a flexible device definition

```
$RTE_TARGET/build/app/testpmd -c 0xff -n 4 -w ff:ff.f \
   --vdev='net_failsafe0,exec(echo 84:00.0)' -- -i
```

5. Start testpmd, automatically probing the device 84:00.0 and using it with the fail-safe.

```
$RTE_TARGET/build/app/testpmd -c 0xff -n 4 \
   --vdev 'net_failsafe0,dev(0000:84:00.0),dev(net_ring0)' -- -i
```

9.55.4 Using the Fail-safe PMD from an application

This driver strives to be as seamless as possible to existing applications, in order to propose the hotplug functionality in the easiest way possible.

Care must be taken, however, to respect the **ether** API concerning device access, and in particular, using the `RTE_ETH_FOREACH_DEV` macro to iterate over ethernet devices, instead of directly accessing them or by writing one's own device iterator.

```
unsigned int i;

/* VALID iteration over eth-dev. */
RTE_ETH_FOREACH_DEV(i) {
        [...]
}

/* INVALID iteration over eth-dev. */
for (i = 0; i < RTE_MAX_ETHPORTS; i++) {
        [...]
}
```

9.55.5 Plug-in feature

A sub-device can be defined without existing on the system when the fail-safe PMD is initialized. Upon probing this device, the fail-safe PMD will detect its absence and postpone its use. It will then register for a periodic check on any missing sub-device.

During this time, the fail-safe PMD can be used normally, configured and told to emit and receive packets. It will store any applied configuration but will fail to emit anything, returning `0` from its TX function. Any unsent packet must be freed.

Upon the probing of its missing sub-device, the current stored configuration will be applied. After this configuration pass, the new sub-device will be synchronized with other sub-devices, i.e. be started if the fail-safe PMD has been started by the user before.

9.55.6 Plug-out feature

A sub-device supporting the device removal event can be removed from its bus at any time. The fail-safe PMD will register a callback for such event and react accordingly. It will try to safely stop, close and uninit the sub-device having emitted this event, allowing it to free its eventual resources.

9.55.7 Fail-safe glossary

Fallback device
: Also called **Secondary device**.

  The fail-safe will fail-over onto this device when the preferred device is absent.

Preferred device
: Also called **Primary device**.

  The first declared sub-device in the fail-safe parameters. When this device is plugged, it is always used as emitting device. It is the main sub-device and is used as target for configuration operations if there is any ambiguity.

Upkeep round
: Periodical event during which sub-devices are serviced. Each devices having a state different to that of the fail-safe device itself, is synchronized with it (brought down or up accordingly). Additionally, any sub-device marked for removal is cleaned-up.

Slave
: In the context of the fail-safe PMD, synonymous to sub-device.

Sub-device

A device being utilized by the fail-safe PMD. This is another PMD running underneath the fail-safe PMD. Any sub-device can disappear at any time. The fail-safe will ensure that the device removal happens gracefully.

CHAPTER

TEN

BASEBAND DEVICE DRIVERS

10.1 Baseband Device Supported Functionality Matrices

10.1.1 Supported Feature Flags

Table 10.1: Features availability in bbdev drivers

Feature	fpga_5gnr_fec	fpga_lte_fec	mbc	null	turbo_sw
Turbo Decoder (4G)		Y	Y		Y
Turbo Encoder (4G)		Y	Y		Y
LDPC Decoder (5G)	Y		Y		Y
LDPC Encoder (5G)	Y		Y		Y
LLR/HARQ Compression			Y		
External DDR Access	Y		Y		
HW Accelerated	Y	Y	Y		
BBDEV API	Y	Y	Y	Y	Y

10.2 BBDEV null Poll Mode Driver

The (**baseband_null**) is a bbdev poll mode driver which provides a minimal implementation of a software bbdev device. As a null device it does not modify the data in the mbuf on which the bbdev operation is to operate and it only works for operation type `RTE_BBDEV_OP_NONE`.

When a burst of mbufs is submitted to a *bbdev null PMD* for processing then each mbuf in the burst will be enqueued in an internal buffer ring to be collected on a dequeue call.

10.2.1 Limitations

- In-place operations for Turbo encode and decode are not supported

10.2.2 Installation

The *bbdev null PMD* is enabled and built by default in both the Linux and FreeBSD builds.

10.2.3 Initialization

To use the PMD in an application, user must:

- Call `rte_vdev_init("baseband_null")` within the application.
- Use `--vdev="baseband_null"` in the EAL options, which will call `rte_vdev_init()` internally.

The following parameters (all optional) can be provided in the previous two calls:

- `socket_id`: Specify the socket where the memory for the device is going to be allocated (by default, *socket_id* will be the socket where the core that is creating the PMD is running on).
- `max_nb_queues`: Specify the maximum number of queues in the device (default is `RTE_MAX_LCORE`).

Example:

```
./test-bbdev.py -e="--vdev=baseband_null,socket_id=0,max_nb_queues=8"
```

10.3 SW Turbo Poll Mode Driver

The SW Turbo PMD (**baseband_turbo_sw**) provides a software only poll mode bbdev driver that can optionally utilize Intel optimized libraries for LTE and 5GNR Layer 1 workloads acceleration.

Note that the driver can also be built without any dependency with reduced functionality for maintenance purpose.

To enable linking to the SDK libraries see detailed installation section below. Two flags can be enabled depending on whether the target machine can support AVX2 and AVX512 instructions sets and the related SDK libraries for vectorized signal processing functions are installed : - CONFIG_RTE_BBDEV_SDK_AVX2 - CONFIG_RTE_BBDEV_SDK_AVX512 By default these 2 flags are disabled by default. For AVX2 machine and SDK library installed then the first flag can be enabled. For AVX512 machine and SDK library installed then both flags can be enabled for full real time capability.

This PMD supports the functions: FEC, Rate Matching and CRC functions detailed in the Features section.

10.3.1 Features

SW Turbo PMD can support for the following capabilities when the SDK libraries are used:

For the LTE encode operation:

- RTE_BBDEV_TURBO_CRC_24A_ATTACH
- RTE_BBDEV_TURBO_CRC_24B_ATTACH
- RTE_BBDEV_TURBO_RATE_MATCH
- RTE_BBDEV_TURBO_RV_INDEX_BYPASS

For the LTE decode operation:

- RTE_BBDEV_TURBO_SUBBLOCK_DEINTERLEAVE
- RTE_BBDEV_TURBO_CRC_TYPE_24B
- RTE_BBDEV_TURBO_POS_LLR_1_BIT_IN
- RTE_BBDEV_TURBO_NEG_LLR_1_BIT_IN
- RTE_BBDEV_TURBO_DEC_TB_CRC_24B_KEEP
- RTE_BBDEV_TURBO_EARLY_TERMINATION

For the 5G NR LDPC encode operation:

- RTE_BBDEV_LDPC_RATE_MATCH
- RTE_BBDEV_LDPC_CRC_24A_ATTACH
- RTE_BBDEV_LDPC_CRC_24B_ATTACH

For the 5G NR LDPC decode operation:

- RTE_BBDEV_LDPC_CRC_TYPE_24B_CHECK
- RTE_BBDEV_LDPC_CRC_TYPE_24A_CHECK
- RTE_BBDEV_LDPC_CRC_TYPE_24B_DROP
- RTE_BBDEV_LDPC_HQ_COMBINE_IN_ENABLE
- RTE_BBDEV_LDPC_HQ_COMBINE_OUT_ENABLE
- RTE_BBDEV_LDPC_ITERATION_STOP_ENABLE

10.3.2 Limitations

- In-place operations for encode and decode are not supported

10.3.3 Installation

FlexRAN SDK Download

As an option it is possible to link this driver with FleXRAN SDK libraries which can enable real time signal processing using AVX instructions.

These libraries are available through this link.

After download is complete, the user needs to unpack and compile on their system before building DPDK.

The following table maps DPDK versions with past FlexRAN SDK releases:

Table 10.2: DPDK and FlexRAN FEC SDK releases compliance

DPDK version	FlexRAN FEC SDK release
19.08	19.04

FlexRAN SDK Installation

Note that the installation of these libraries is optional.

The following are pre-requisites for building FlexRAN SDK Libraries:

(a) An AVX2 or AVX512 supporting machine

(b) CentOS Linux release 7.2.1511 (Core) operating system is advised

(c) Intel ICC 18.0.1 20171018 compiler or more recent and related libraries ICC is available with a free community license.

The following instructions should be followed in this exact order:

1. Set the environment variables:

```
source <path-to-icc-compiler-install-folder>/linux/bin/compilervars.sh intel64␣
↪-platform linux
```

2. Run the SDK extractor script and accept the license:

```
cd <path-to-workspace>
./FlexRAN-FEC-SDK-19-04.sh
```

3. Generate makefiles based on system configuration:

```
cd <path-to-workspace>/FlexRAN-FEC-SDK-19-04/sdk/
./create-makefiles-linux.sh
```

4. A build folder is generated in this form `build-<ISA>-<CC>`, enter that folder and install:

```
cd build-avx512-icc/
make && make install
```

10.3.4 Initialization

In order to enable this virtual bbdev PMD, the user may:

- Build the FLEXRAN SDK libraries (explained in Installation section).
- Export the environmental variables FLEXRAN_SDK to the path where the FlexRAN SDK libraries were installed. And DIR_WIRELESS_SDK to the path where the libraries were extracted.

Example:

```
export FLEXRAN_SDK=<path-to-workspace>/FlexRAN-FEC-SDK-19-04/sdk/build-avx2-icc/install
export DIR_WIRELESS_SDK=<path-to-workspace>/FlexRAN-FEC-SDK-19-04/sdk/build-avx2-icc/
```

- Set CONFIG_RTE_BBDEV_SDK_AVX2=y and CONFIG_RTE_BBDEV_SDK_AVX512=y in DPDK common configuration file config/common_base to be able to use the SDK libraries as mentioned above. For AVX2 machine it is possible to only enable CONFIG_RTE_BBDEV_SDK_AVX2 for limited 4G functionality. If no flag are set the PMD driver will still build but its capabilities will be limited accordingly.

To use the PMD in an application, user must:

- Call rte_vdev_init("baseband_turbo_sw") within the application.
- Use --vdev="baseband_turbo_sw" in the EAL options, which will call rte_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, *socket_id* will be the socket where the core that is creating the PMD is running on).
- max_nb_queues: Specify the maximum number of queues in the device (default is RTE_MAX_LCORE).

Example:

```
./test-bbdev.py -e="--vdev=baseband_turbo_sw,socket_id=0,max_nb_queues=8" \
-c validation -v ./turbo_*_default.data
```

10.4 Intel(R) FPGA LTE FEC Poll Mode Driver

The BBDEV FPGA LTE FEC poll mode driver (PMD) supports an FPGA implementation of a VRAN Turbo Encode / Decode LTE wireless acceleration function, using Intel's PCI-e and FPGA based Vista Creek device.

10.4.1 Features

FPGA LTE FEC PMD supports the following features:

- Turbo Encode in the DL with total throughput of 4.5 Gbits/s
- Turbo Decode in the UL with total throughput of 1.5 Gbits/s assuming 8 decoder iterations
- 8 VFs per PF (physical device)
- Maximum of 32 UL queues per VF
- Maximum of 32 DL queues per VF
- PCIe Gen-3 x8 Interface
- MSI-X
- SR-IOV

FPGA LTE FEC PMD supports the following BBDEV capabilities:

- **For the turbo encode operation:**
  - `RTE_BBDEV_TURBO_CRC_24B_ATTACH` : set to attach CRC24B to CB(s)
  - `RTE_BBDEV_TURBO_RATE_MATCH` : if set then do not do Rate Match bypass
  - `RTE_BBDEV_TURBO_ENC_INTERRUPTS` : set for encoder dequeue interrupts
- **For the turbo decode operation:**
  - `RTE_BBDEV_TURBO_CRC_TYPE_24B` : check CRC24B from CB(s)
  - `RTE_BBDEV_TURBO_SUBBLOCK_DEINTERLEAVE` : perform subblock de-interleave
  - `RTE_BBDEV_TURBO_DEC_INTERRUPTS` : set for decoder dequeue interrupts
  - `RTE_BBDEV_TURBO_NEG_LLR_1_BIT_IN` : set if negative LLR encoder i/p is supported
  - `RTE_BBDEV_TURBO_DEC_TB_CRC_24B_KEEP` : keep CRC24B bits appended while decoding

10.4.2 Limitations

FPGA LTE FEC does not support the following:

- Scatter-Gather function

10.4.3 Installation

Section 3 of the DPDK manual provides instuctions on installing and compiling DPDK. The default set of bbdev compile flags may be found in config/common_base, where for example the flag to build the FPGA LTE FEC device, `CONFIG_RTE_LIBRTE_PMD_BBDEV_FPGA_LTE_FEC`, is already set. It is assumed DPDK has been compiled using for instance:

```
make install T=x86_64-native-linuxapp-gcc
```

DPDK requires hugepages to be configured as detailed in section 2 of the DPDK manual. The bbdev test application has been tested with a configuration 40 x 1GB hugepages. The hugepage configuration of a server may be examined using:

```
grep Huge* /proc/meminfo
```

10.4.4 Initialization

When the device first powers up, its PCI Physical Functions (PF) can be listed through this command:

```
sudo lspci -vd1172:5052
```

The physical and virtual functions are compatible with Linux UIO drivers: `vfio` and `igb_uio`. However, in order to work the FPGA LTE FEC device firstly needs to be bound to one of these linux drivers through DPDK.

Bind PF UIO driver(s)

Install the DPDK igb_uio driver, bind it with the PF PCI device ID and use `lspci` to confirm the PF device is under use by `igb_uio` DPDK UIO driver.

The igb_uio driver may be bound to the PF PCI device using one of three methods:

1. PCI functions (physical or virtual, depending on the use case) can be bound to the UIO driver by repeating this command for every function.

```
cd <dpdk-top-level-directory>
insmod ./build/kmod/igb_uio.ko
echo "1172 5052" > /sys/bus/pci/drivers/igb_uio/new_id
lspci -vd1172:
```

2. Another way to bind PF with DPDK UIO driver is by using the `dpdk-devbind.py` tool

```
cd <dpdk-top-level-directory>
./usertools/dpdk-devbind.py -b igb_uio 0000:06:00.0
```

where the PCI device ID (example: 0000:06:00.0) is obtained using lspci -vd1172:

3. A third way to bind is to use `dpdk-setup.sh` tool

```
cd <dpdk-top-level-directory>
./usertools/dpdk-setup.sh

select 'Bind Ethernet/Crypto/Baseband device to IGB UIO module'
or
select 'Bind Ethernet/Crypto/Baseband device to VFIO module' depending on driver required
enter PCI device ID
select 'Display current Ethernet/Crypto/Baseband device settings' to confirm binding
```

In the same way the FPGA LTE FEC PF can be bound with vfio, but vfio driver does not support SR-IOV configuration right out of the box, so it will need to be patched.

Enable Virtual Functions

Now, it should be visible in the printouts that PCI PF is under igb_uio control "`Kernel driver in use: igb_uio`"

To show the number of available VFs on the device, read `sriov_totalvfs` file..

```
cat /sys/bus/pci/devices/0000\:<b>\:<d>.<f>/sriov_totalvfs

where 0000\:<b>\:<d>.<f> is the PCI device ID
```

To enable VFs via igb_uio, echo the number of virtual functions intended to enable to `max_vfs` file..

```
echo <num-of-vfs> > /sys/bus/pci/devices/0000\:<b>\:<d>.<f>/max_vfs
```

Afterwards, all VFs must be bound to appropriate UIO drivers as required, same way it was done with the physical function previously.

Enabling SR-IOV via vfio driver is pretty much the same, except that the file name is different:

```
echo <num-of-vfs> > /sys/bus/pci/devices/0000\:<b>\:<d>.<f>/sriov_numvfs
```

Configure the VFs through PF

The PCI virtual functions must be configured before working or getting assigned to VMs/Containers. The configuration involves allocating the number of hardware queues, priorities, load balance, bandwidth and other settings necessary for the device to perform FEC functions.

This configuration needs to be executed at least once after reboot or PCI FLR and can be achieved by using the function `fpga_lte_fec_configure()`, which sets up the parameters defined in `fpga_lte_fec_conf` structure:

```
struct fpga_lte_fec_conf {
    bool pf_mode_en;
    uint8_t vf_ul_queues_number[FPGA_LTE_FEC_NUM_VFS];
    uint8_t vf_dl_queues_number[FPGA_LTE_FEC_NUM_VFS];
    uint8_t ul_bandwidth;
    uint8_t dl_bandwidth;
    uint8_t ul_load_balance;
    uint8_t dl_load_balance;
    uint16_t flr_time_out;
};
```

- `pf_mode_en`: identifies whether only PF is to be used, or the VFs. PF and VFs are mutually exclusive and cannot run simultaneously. Set to 1 for PF mode enabled. If PF mode is enabled all queues available in the device are assigned exclusively to PF and 0 queues given to VFs.
- `vf_*l_queues_number`: defines the hardware queue mapping for every VF.
- `*l_bandwidth`: in case of congestion on PCIe interface. The device allocates different bandwidth to UL and DL. The weight is configured by this setting. The unit of weight is 3 code blocks. For example, if the code block cbps (code block per second) ratio between UL and DL is 12:1, then the configuration value should be set to 36:3. The schedule algorithm is based on code block regardless the length of each block.

- *l_load_balance: hardware queues are load-balanced in a round-robin fashion. Queues get filled first-in first-out until they reach a pre-defined watermark level, if exceeded, they won't get assigned new code blocks.. This watermark is defined by this setting.

  If all hardware queues exceeds the watermark, no code blocks will be streamed in from UL/DL code block FIFO.

- flr_time_out: specifies how many 16.384us to be FLR time out. The time_out = flr_time_out x 16.384us. For instance, if you want to set 10ms for the FLR time out then set this setting to 0x262=610.

An example configuration code calling the function fpga_lte_fec_configure() is shown below:

```
struct fpga_lte_fec_conf conf;
unsigned int i;

memset(&conf, 0, sizeof(struct fpga_lte_fec_conf));
conf.pf_mode_en = 1;

for (i = 0; i < FPGA_LTE_FEC_NUM_VFS; ++i) {
    conf.vf_ul_queues_number[i] = 4;
    conf.vf_dl_queues_number[i] = 4;
}
conf.ul_bandwidth = 12;
conf.dl_bandwidth = 5;
conf.dl_load_balance = 64;
conf.ul_load_balance = 64;

/* setup FPGA PF */
ret = fpga_lte_fec_configure(info->dev_name, &conf);
TEST_ASSERT_SUCCESS(ret,
    "Failed to configure 4G FPGA PF for bbdev %s",
    info->dev_name);
```

10.4.5 Test Application

BBDEV provides a test application, test-bbdev.py and range of test data for testing the functionality of FPGA LTE FEC turbo encode and turbo decode, depending on the device's capabilities. The test application is located under app->test-bbdev folder and has the following options:

```
"-p", "--testapp-path": specifies path to the bbdev test app.
"-e", "--eal-params"  : EAL arguments which are passed to the test app.
"-t", "--timeout"     : Timeout in seconds (default=300).
"-c", "--test-cases"  : Defines test cases to run. Run all if not specified.
"-v", "--test-vector" : Test vector path (default=dpdk_path+/app/test-bbdev/test_vectors/bbdev_
↪null.data).
"-n", "--num-ops"     : Number of operations to process on device (default=32).
"-b", "--burst-size"  : Operations enqueue/dequeue burst size (default=32).
"-l", "--num-lcores"  : Number of lcores to run (default=16).
"-i", "--init-device" : Initialise PF device with default values.
```

To execute the test application tool using simple turbo decode or turbo encode data, type one of the following:

```
./test-bbdev.py -c validation -n 64 -b 8 -v ./turbo_dec_default.data
./test-bbdev.py -c validation -n 64 -b 8 -v ./turbo_enc_default.data
```

The test application test-bbdev.py, supports the ability to configure the PF device with a default

set of values, if the "-i" or "- -init-device" option is included. The default values are defined in test_bbdev_perf.c as:

- VF_UL_QUEUE_VALUE 4
- VF_DL_QUEUE_VALUE 4
- UL_BANDWIDTH 3
- DL_BANDWIDTH 3
- UL_LOAD_BALANCE 128
- DL_LOAD_BALANCE 128
- FLR_TIMEOUT 610

Test Vectors

In addition to the simple turbo decoder and turbo encoder tests, bbdev also provides a range of additional tests under the test_vectors folder, which may be useful. The results of these tests will depend on the FPGA LTE FEC capabilities:

- **turbo decoder tests:**
  - `turbo_dec_c1_k6144_r0_e10376_crc24b_sbd_negllr_high_snr.data`
  - `turbo_dec_c1_k6144_r0_e10376_crc24b_sbd_negllr_low_snr.data`
  - `turbo_dec_c1_k6144_r0_e34560_negllr.data`
  - `turbo_dec_c1_k6144_r0_e34560_sbd_negllr.data`
  - `turbo_dec_c2_k3136_r0_e4920_sbd_negllr_crc24b.data`
  - `turbo_dec_c2_k3136_r0_e4920_sbd_negllr.data`
- **turbo encoder tests:**
  - `turbo_enc_c1_k40_r0_e1190_rm.data`
  - `turbo_enc_c1_k40_r0_e1194_rm.data`
  - `turbo_enc_c1_k40_r0_e1196_rm.data`
  - `turbo_enc_c1_k40_r0_e272_rm.data`
  - `turbo_enc_c1_k6144_r0_e18444.data`
  - `turbo_enc_c1_k6144_r0_e32256_crc24b_rm.data`
  - `turbo_enc_c2_k5952_r0_e17868_crc24b.data`
  - `turbo_enc_c3_k4800_r2_e14412_crc24b.data`
  - `turbo_enc_c4_k4800_r2_e14412_crc24b.data`

10.5 Intel(R) FPGA 5GNR FEC Poll Mode Driver

The BBDEV FPGA 5GNR FEC poll mode driver (PMD) supports an FPGA implementation of a VRAN LDPC Encode / Decode 5GNR wireless acceleration function, using Intel's PCI-e and FPGA based Vista Creek device.

10.5.1 Features

FPGA 5GNR FEC PMD supports the following features:

- LDPC Encode in the DL
- LDPC Decode in the UL
- 8 VFs per PF (physical device)
- Maximum of 32 UL queues per VF
- Maximum of 32 DL queues per VF
- PCIe Gen-3 x8 Interface
- MSI-X
- SR-IOV

FPGA 5GNR FEC PMD supports the following BBDEV capabilities:

- **For the LDPC encode operation:**
  - `RTE_BBDEV_LDPC_CRC_24B_ATTACH` : set to attach CRC24B to CB(s)
  - `RTE_BBDEV_LDPC_RATE_MATCH` : if set then do not do Rate Match bypass
- **For the LDPC decode operation:**
  - `RTE_BBDEV_LDPC_CRC_TYPE_24B_CHECK` : check CRC24B from CB(s)
  - `RTE_BBDEV_LDPC_ITERATION_STOP_ENABLE` : disable early termination
  - `RTE_BBDEV_LDPC_CRC_TYPE_24B_DROP` : drops CRC24B bits appended while decoding
  - `RTE_BBDEV_LDPC_HQ_COMBINE_IN_ENABLE` : provides an input for HARQ combining
  - `RTE_BBDEV_LDPC_HQ_COMBINE_OUT_ENABLE` : provides an input for HARQ combining
  - `RTE_BBDEV_LDPC_INTERNAL_HARQ_MEMORY_IN_ENABLE` : HARQ memory input is internal
  - `RTE_BBDEV_LDPC_INTERNAL_HARQ_MEMORY_OUT_ENABLE` : HARQ memory output is internal
  - `RTE_BBDEV_LDPC_INTERNAL_HARQ_MEMORY_LOOPBACK` : loopback data to/from HARQ memory
  - `RTE_BBDEV_LDPC_INTERNAL_HARQ_MEMORY_FILLERS` : HARQ memory includes the fillers bits

10.5.2 Limitations

FPGA 5GNR FEC does not support the following:

- Scatter-Gather function

10.5.3 Installation

Section 3 of the DPDK manual provides instuctions on installing and compiling DPDK. The default set of bbdev compile flags may be found in config/common_base, where for example the flag to build the FPGA 5GNR FEC device, `CONFIG_RTE_LIBRTE_PMD_BBDEV_FPGA_5GNR_FEC`, is already set. It is assumed DPDK has been compiled using for instance:

```
make install T=x86_64-native-linuxapp-gcc
```

DPDK requires hugepages to be configured as detailed in section 2 of the DPDK manual. The bbdev test application has been tested with a configuration 40 x 1GB hugepages. The hugepage configuration of a server may be examined using:

```
grep Huge* /proc/meminfo
```

10.5.4 Initialization

When the device first powers up, its PCI Physical Functions (PF) can be listed through this command:

```
sudo lspci -vd8086:0d8f
```

The physical and virtual functions are compatible with Linux UIO drivers: `vfio` and `igb_uio`. However, in order to work the FPGA 5GNR FEC device firstly needs to be bound to one of these linux drivers through DPDK.

Bind PF UIO driver(s)

Install the DPDK igb_uio driver, bind it with the PF PCI device ID and use `lspci` to confirm the PF device is under use by `igb_uio` DPDK UIO driver.

The igb_uio driver may be bound to the PF PCI device using one of three methods:

1. PCI functions (physical or virtual, depending on the use case) can be bound to the UIO driver by repeating this command for every function.

```
cd <dpdk-top-level-directory>
insmod ./build/kmod/igb_uio.ko
echo "8086 0d8f" > /sys/bus/pci/drivers/igb_uio/new_id
lspci -vd8086:0d8f
```

2. Another way to bind PF with DPDK UIO driver is by using the `dpdk-devbind.py` tool

```
cd <dpdk-top-level-directory>
./usertools/dpdk-devbind.py -b igb_uio 0000:06:00.0
```

where the PCI device ID (example: 0000:06:00.0) is obtained using lspci -vd8086:0d8f

3. A third way to bind is to use `dpdk-setup.sh` tool

```
cd <dpdk-top-level-directory>
./usertools/dpdk-setup.sh

select 'Bind Ethernet/Crypto/Baseband device to IGB UIO module'
or
select 'Bind Ethernet/Crypto/Baseband device to VFIO module' depending on driver required
enter PCI device ID
select 'Display current Ethernet/Crypto/Baseband device settings' to confirm binding
```

In the same way the FPGA 5GNR FEC PF can be bound with vfio, but vfio driver does not support SR-IOV configuration right out of the box, so it will need to be patched.

Enable Virtual Functions

Now, it should be visible in the printouts that PCI PF is under igb_uio control "`Kernel driver in use: igb_uio`"

To show the number of available VFs on the device, read `sriov_totalvfs` file..

```
cat /sys/bus/pci/devices/0000\:<b>\:<d>.<f>/sriov_totalvfs

where 0000\:<b>\:<d>.<f> is the PCI device ID
```

To enable VFs via igb_uio, echo the number of virtual functions intended to enable to `max_vfs` file..

```
echo <num-of-vfs> > /sys/bus/pci/devices/0000\:<b>\:<d>.<f>/max_vfs
```

Afterwards, all VFs must be bound to appropriate UIO drivers as required, same way it was done with the physical function previously.

Enabling SR-IOV via vfio driver is pretty much the same, except that the file name is different:

```
echo <num-of-vfs> > /sys/bus/pci/devices/0000\:<b>\:<d>.<f>/sriov_numvfs
```

Configure the VFs through PF

The PCI virtual functions must be configured before working or getting assigned to VMs/Containers. The configuration involves allocating the number of hardware queues, priorities, load balance, bandwidth and other settings necessary for the device to perform FEC functions.

This configuration needs to be executed at least once after reboot or PCI FLR and can be achieved by using the function `fpga_5gnr_fec_configure()`, which sets up the parameters defined in `fpga_5gnr_fec_conf` structure:

```
struct fpga_5gnr_fec_conf {
    bool pf_mode_en;
    uint8_t vf_ul_queues_number[FPGA_5GNR_FEC_NUM_VFS];
    uint8_t vf_dl_queues_number[FPGA_5GNR_FEC_NUM_VFS];
    uint8_t ul_bandwidth;
    uint8_t dl_bandwidth;
    uint8_t ul_load_balance;
    uint8_t dl_load_balance;
    uint16_t flr_time_out;
};
```

- `pf_mode_en`: identifies whether only PF is to be used, or the VFs. PF and VFs are mutually exclusive and cannot run simultaneously. Set to 1 for PF mode enabled. If PF mode is enabled all queues available in the device are assigned exclusively to PF and 0 queues given to VFs.
- `vf_*l_queues_number`: defines the hardware queue mapping for every VF.
- `*l_bandwidth`: in case of congestion on PCIe interface. The device allocates different bandwidth to UL and DL. The weight is configured by this setting. The unit of weight is 3 code blocks. For example, if the code block cbps (code block per second) ratio between UL and DL is 12:1, then the configuration value should be set to 36:3. The schedule algorithm is based on code block regardless the length of each block.
- `*l_load_balance`: hardware queues are load-balanced in a round-robin fashion. Queues get filled first-in first-out until they reach a pre-defined watermark level, if exceeded, they won't get assigned new code blocks.. This watermark is defined by this setting.

  If all hardware queues exceeds the watermark, no code blocks will be streamed in from UL/DL code block FIFO.
- `flr_time_out`: specifies how many 16.384us to be FLR time out. The time_out = flr_time_out x 16.384us. For instance, if you want to set 10ms for the FLR time out then set this setting to 0x262=610.

An example configuration code calling the function `fpga_5gnr_fec_configure()` is shown below:

```
struct fpga_5gnr_fec_conf conf;
unsigned int i;

memset(&conf, 0, sizeof(struct fpga_5gnr_fec_conf));
conf.pf_mode_en = 1;

for (i = 0; i < FPGA_5GNR_FEC_NUM_VFS; ++i) {
    conf.vf_ul_queues_number[i] = 4;
    conf.vf_dl_queues_number[i] = 4;
}
conf.ul_bandwidth = 12;
conf.dl_bandwidth = 5;
conf.dl_load_balance = 64;
conf.ul_load_balance = 64;

/* setup FPGA PF */
ret = fpga_5gnr_fec_configure(info->dev_name, &conf);
TEST_ASSERT_SUCCESS(ret,
    "Failed to configure 4G FPGA PF for bbdev %s",
    info->dev_name);
```

10.5.5 Test Application

BBDEV provides a test application, `test-bbdev.py` and range of test data for testing the functionality of FPGA 5GNR FEC encode and decode, depending on the device's capabilities. The test application is located under app->test-bbdev folder and has the following options:

```
"-p", "--testapp-path": specifies path to the bbdev test app.
"-e", "--eal-params"  : EAL arguments which are passed to the test app.
"-t", "--timeout"     : Timeout in seconds (default=300).
"-c", "--test-cases"  : Defines test cases to run. Run all if not specified.
"-v", "--test-vector" : Test vector path (default=dpdk_path+/app/test-bbdev/test_vectors/bbdev_
↪null.data).
```

(continues on next page)

(continued from previous page)

```
"-n", "--num-ops"     : Number of operations to process on device (default=32).
"-b", "--burst-size"  : Operations enqueue/dequeue burst size (default=32).
"-l", "--num-lcores"  : Number of lcores to run (default=16).
"-i", "--init-device" : Initialise PF device with default values.
```

To execute the test application tool using simple decode or encode data, type one of the following:

```
./test-bbdev.py -c validation -n 64 -b 1 -v ./ldpc_dec_default.data
./test-bbdev.py -c validation -n 64 -b 1 -v ./ldpc_enc_default.data
```

The test application `test-bbdev.py`, supports the ability to configure the PF device with a default set of values, if the "-i" or "- -init-device" option is included. The default values are defined in test_bbdev_perf.c as:

- VF_UL_QUEUE_VALUE 4
- VF_DL_QUEUE_VALUE 4
- UL_BANDWIDTH 3
- DL_BANDWIDTH 3
- UL_LOAD_BALANCE 128
- DL_LOAD_BALANCE 128
- FLR_TIMEOUT 610

Test Vectors

In addition to the simple LDPC decoder and LDPC encoder tests, bbdev also provides a range of additional tests under the test_vectors folder, which may be useful. The results of these tests will depend on the FPGA 5GNR FEC capabilities.

CHAPTER

ELEVEN

CRYPTO DEVICE DRIVERS

11.1 Crypto Device Supported Functionality Matrices

11.1.1 Supported Feature Flags

Table 11.1: Features availability in crypto drivers

Feature	aesni_gcm	aesni_mb	armv8	caam_jr	ccp	dpaa2_sec	dpaa_sec	kasumi	mvsam	nitrox	null	octeontx	octeontx2	openssl	qat	snow3g	virtio	zuc
Symmetric crypto	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Asymmetric crypto												Y	Y	Y				
Sym operation chaining	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
HW Accelerated				Y	Y	Y	Y		Y	Y		Y	Y		Y			
Protocol offload				Y		Y	Y											
CPU SSE	Y	Y																
CPU AVX	Y	Y																

Note:

- "In Place SGL" feature flag stands for "In place Scatter-gather list", which means that an input buffer can consist of multiple segments, being the operation in-place (input address = output address).
- "OOP SGL In SGL Out" feature flag stands for "Out-of-place Scatter-gather list Input, Scatter-gather list Output", which means pmd supports different scatter-gather styled input and output buffers (i.e. both can consists of multiple segments).
- "OOP SGL In LB Out" feature flag stands for "Out-of-place Scatter-gather list Input, Linear Buffers Output", which means PMD supports input from scatter-gathered styled buffers, outputting linear buffers (i.e. single segment).
- "OOP LB In SGL Out" feature flag stands for "Out-of-place Linear Buffers Input, Scatter-gather list Output", which means PMD supports input from linear buffer, outputting scatter-gathered styled buffers.
- "OOP LB In LB Out" feature flag stands for "Out-of-place Linear Buffers Input, Linear Buffers Output", which means that Out-of-place operation is supported, with linear input and output buffers.
- "RSA PRIV OP KEY EXP" feature flag means PMD support RSA private key operation (Sign and Decrypt) using exponent key type only.
- "RSA PRIV OP KEY QT" feature flag means PMD support RSA private key operation (Sign and Decrypt) using quintuple (crt) type key only.
- "Digest encrypted" feature flag means PMD support hash-cipher cases, where generated digest is appended to and encrypted with the data.

11.1.2 Supported Cipher Algorithms

Table 11.2: Cipher algorithms in crypto drivers

Cipher algorithm	aesni_gcm	aesni_mb	armv8	caam_jr	ccp	dpaa2_sec	dpaa_sec	kasumi	mvsam	nitrox	null	octeontx	octeontx2	openssl	qat	snow3g	virtio	zuc
NULL									Y		Y	Y	Y		Y			
AES CBC (128)	Y	Y	Y	Y	Y	Y			Y	Y		Y	Y	Y	Y		Y	
AES CBC (192)	Y		Y	Y	Y	Y			Y	Y		Y	Y	Y	Y		Y	
AES CBC (256)	Y		Y	Y	Y	Y			Y	Y		Y	Y	Y	Y		Y	
AES ECB (128)				Y					Y									
AES ECB (192)				Y					Y									
AES ECB (256)				Y					Y									
AES CTR (128)	Y		Y	Y	Y	Y			Y			Y	Y	Y	Y			
AES CTR (192)	Y		Y	Y	Y	Y			Y			Y	Y	Y	Y			
AES CTR (256)	Y		Y	Y	Y	Y			Y			Y	Y	Y	Y			
AES XTS (128)												Y	Y		Y			
AES XTS (192)																		
AES XTS (256)												Y	Y		Y			
AES DOCSIS BPI	Y														Y			
3DES CBC	Y		Y	Y	Y	Y			Y	Y		Y	Y	Y	Y			

11.1.3 Supported Authentication Algorithms

Table 11.3: Authentication algorithms in crypto drivers

Authentication algorithm	aesni_gcm	aesni_mb	armv8	caam_jr	ccp	dpaa2_sec	dpaa_sec	kasumi	mvsam	nitrox	null	octeontx	octeontx2	openssl	qat	snow3g	virtio	zuc
NULL									Y		Y	Y	Y		Y			
MD5									Y			Y	Y	Y				
MD5 HMAC	Y		Y	Y	Y	Y			Y			Y	Y	Y	Y			
SHA1	Y			Y					Y			Y	Y	Y	Y			
SHA1 HMAC	Y	Y	Y	Y	Y	Y			Y	Y		Y	Y	Y	Y		Y	
SHA224	Y			Y					Y			Y	Y	Y	Y			
SHA224 HMAC	Y		Y	Y	Y	Y			Y	Y		Y	Y	Y	Y			
SHA256	Y			Y					Y			Y	Y	Y	Y			
SHA256 HMAC	Y	Y	Y	Y	Y	Y			Y	Y		Y	Y	Y	Y			
SHA384	Y			Y					Y			Y	Y	Y	Y			
SHA384 HMAC	Y		Y	Y	Y	Y			Y			Y	Y	Y	Y			
SHA512	Y			Y					Y			Y	Y	Y	Y			
SHA512 HMAC	Y		Y	Y	Y	Y			Y			Y	Y	Y	Y			
AES XCBC MAC	Y														Y			
AES GMAC	Y								Y			Y	Y	Y	Y			
SNOW3G UIA2					Y	Y						Y	Y		Y	Y		
KASUMI F9							Y					Y	Y		Y			
ZUC EIA3					Y	Y						Y	Y		Y			Y
AES CMAC (128)	Y			Y											Y			
AES CMAC (192)				Y														
AES CMAC (256)				Y														
SHA3_224				Y														
SHA3_224 HMAC				Y														

11.1.4 Supported AEAD Algorithms

Table 11.4: AEAD algorithms in crypto drivers

AEAD algorithm	aesni_gcm	aesni_mb	armv8	caam_jr	ccp	dpaa2_sec	dpaa_sec	kasumi	mvsam	nitrox	null	octeontx	octeontx2	openssl	qat	snow3g	virtio	zuc
AES GCM (128)	Y	Y		Y	Y	Y	Y		Y			Y	Y	Y	Y			
AES GCM (192)	Y	Y		Y	Y	Y	Y		Y			Y	Y	Y	Y			
AES GCM (256)	Y	Y		Y	Y	Y	Y		Y			Y	Y	Y	Y			
AES CCM (128)		Y												Y	Y			
AES CCM (192)														Y	Y			
AES CCM (256)														Y	Y			
CHACHA20-POLY1305																		

11.1.5 Supported Asymmetric Algorithms

Table 11.5: Asymmetric algorithms in crypto drivers

Asymmetric algorithm	aesni_gcm	aesni_mb	armv8	caam_jr	ccp	dpaa2_sec	dpaa_sec	kasumi	mvsam	nitrox	null	octeontx	octeontx2	openssl	qat	snow3g	virtio	zuc
RSA												Y	Y	Y	Y			
DSA														Y				
Modular Exponentiation												Y	Y	Y	Y			
Modular Inversion														Y	Y			
Diffie-hellman														Y				
ECDSA												Y	Y					
ECPM												Y	Y					

11.2 AESN-NI Multi Buffer Crypto Poll Mode Driver

The AESNI MB PMD (**librte_pmd_aesni_mb**) provides poll mode crypto driver support for utilizing Intel multi buffer library, see the white paper Fast Multi-buffer IPsec Implementations on Intel® Architecture Processors.

The AES-NI MB PMD has current only been tested on Fedora 21 64-bit with gcc.

The AES-NI MB PMD supports synchronous mode of operation with `rte_cryptodev_sym_cpu_crypto_process` function call.

11.2.1 Features

AESNI MB PMD has support for:

Cipher algorithms:

- RTE_CRYPTO_CIPHER_AES128_CBC
- RTE_CRYPTO_CIPHER_AES192_CBC
- RTE_CRYPTO_CIPHER_AES256_CBC
- RTE_CRYPTO_CIPHER_AES128_CTR
- RTE_CRYPTO_CIPHER_AES192_CTR
- RTE_CRYPTO_CIPHER_AES256_CTR
- RTE_CRYPTO_CIPHER_AES_DOCSISBPI
- RTE_CRYPTO_CIPHER_DES_CBC
- RTE_CRYPTO_CIPHER_3DES_CBC
- RTE_CRYPTO_CIPHER_DES_DOCSISBPI

Hash algorithms:

- RTE_CRYPTO_HASH_MD5_HMAC
- RTE_CRYPTO_HASH_SHA1_HMAC
- RTE_CRYPTO_HASH_SHA224_HMAC
- RTE_CRYPTO_HASH_SHA256_HMAC
- RTE_CRYPTO_HASH_SHA384_HMAC
- RTE_CRYPTO_HASH_SHA512_HMAC
- RTE_CRYPTO_HASH_AES_XCBC_HMAC
- RTE_CRYPTO_HASH_AES_CMAC
- RTE_CRYPTO_HASH_AES_GMAC
- RTE_CRYPTO_HASH_SHA1
- RTE_CRYPTO_HASH_SHA224
- RTE_CRYPTO_HASH_SHA256
- RTE_CRYPTO_HASH_SHA384
- RTE_CRYPTO_HASH_SHA512

AEAD algorithms:

- RTE_CRYPTO_AEAD_AES_CCM
- RTE_CRYPTO_AEAD_AES_GCM

11.2.2 Limitations

- Chained mbufs are not supported.

11.2.3 Installation

To build DPDK with the AESNI_MB_PMD the user is required to download the multi-buffer library from here and compile it on their user system before building DPDK. The latest version of the library supported by this PMD is v0.54, which can be downloaded from https://github.com/01org/intel-ipsec-mb/archive/v0.54.zip.

```
make
make install
```

The library requires NASM to be built. Depending on the library version, it might require a minimum NASM version (e.g. v0.54 requires at least NASM 2.14).

NASM is packaged for different OS. However, on some OS the version is too old, so a manual installation is required. In that case, NASM can be downloaded from NASM website. Once it is downloaded, extract it and follow these steps:

```
./configure
make
make install
```

Note: Compilation of the Multi-Buffer library is broken when GCC < 5.0, if library <= v0.53. If a lower GCC version than 5.0, the workaround proposed by the following link should be used: https://github.com/intel/intel-ipsec-mb/issues/40.

As a reference, the following table shows a mapping between the past DPDK versions and the Multi-Buffer library version supported by them:

Table 11.6: DPDK and Multi-Buffer library version compatibility

DPDK version	Multi-buffer library version
2.2 - 16.11	0.43 - 0.44
17.02	0.44
17.05 - 17.08	0.45 - 0.48
17.11	0.47 - 0.48
18.02	0.48
18.05 - 19.02	0.49 - 0.52
19.05 - 19.08	0.52
19.11+	0.52 - 0.54

11.2.4 Initialization

In order to enable this virtual crypto PMD, user must:

- Build the multi buffer library (explained in Installation section).
- Set CONFIG_RTE_LIBRTE_PMD_AESNI_MB=y in config/common_base.

To use the PMD in an application, user must:

- Call rte_vdev_init("crypto_aesni_mb") within the application.
- Use –vdev="crypto_aesni_mb" in the EAL options, which will call rte_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 1 -n 4 --vdev="crypto_aesni_mb,socket_id=0,max_nb_sessions=128" \
-- -p 1 --cdev SW --chain CIPHER_HASH --cipher_algo "aes-cbc" --auth_algo "sha1-hmac"
```

11.2.5 Extra notes

For AES Counter mode (AES-CTR), the library supports two different sizes for Initialization Vector (IV):

- 12 bytes: used mainly for IPsec, as it requires 12 bytes from the user, which internally are appended the counter block (4 bytes), which is set to 1 for the first block (no padding required from the user)
- 16 bytes: when passing 16 bytes, the library will take them and use the last 4 bytes as the initial counter block for the first block.

11.3 AES-NI GCM Crypto Poll Mode Driver

The AES-NI GCM PMD (**librte_pmd_aesni_gcm**) provides poll mode crypto driver support for utilizing Intel multi buffer library (see AES-NI Multi-buffer PMD documentation to learn more about it, including installation).

The AES-NI GCM PMD supports synchronous mode of operation with `rte_cryptodev_sym_cpu_crypto_process` function call for both AES-GCM and GMAC, however GMAC support is limited to one segment per operation. Please refer to `rte_crypto` programmer's guide for more detail.

11.3.1 Features

AESNI GCM PMD has support for:

Authentication algorithms:

- RTE_CRYPTO_AUTH_AES_GMAC

AEAD algorithms:

- RTE_CRYPTO_AEAD_AES_GCM

11.3.2 Limitations

- In out-of-place operations, chained destination mbufs are not supported.
- Chained mbufs are only supported by RTE_CRYPTO_AEAD_AES_GCM algorithm, not RTE_CRYPTO_AUTH_AES_GMAC.
- Cipher only is not supported.

11.3.3 Installation

To build DPDK with the AESNI_GCM_PMD the user is required to download the multi-buffer library from here and compile it on their user system before building DPDK. The latest version of the library supported by this PMD is v0.54, which can be downloaded in https://github.com/01org/intel-ipsec-mb/archive/v0.54.zip.

```
make
make install
```

The library requires NASM to be built. Depending on the library version, it might require a minimum NASM version (e.g. v0.54 requires at least NASM 2.14).

NASM is packaged for different OS. However, on some OS the version is too old, so a manual installation is required. In that case, NASM can be downloaded from NASM website. Once it is downloaded, extract it and follow these steps:

```
./configure
make
make install
```

Note: Compilation of the Multi-Buffer library is broken when GCC < 5.0, if library <= v0.53. If a lower GCC version than 5.0, the workaround proposed by the following link should be used: https://github.com/intel/intel-ipsec-mb/issues/40.

As a reference, the following table shows a mapping between the past DPDK versions and the external crypto libraries supported by them:

Table 11.7: DPDK and external crypto library version compatibility

DPDK version	Crypto library version
16.04 - 16.11	Multi-buffer library 0.43 - 0.44
17.02 - 17.05	ISA-L Crypto v2.18
17.08 - 18.02	Multi-buffer library 0.46 - 0.48
18.05 - 19.02	Multi-buffer library 0.49 - 0.52
19.05+	Multi-buffer library 0.52 - 0.54

11.3.4 Initialization

In order to enable this virtual crypto PMD, user must:

- Build the multi buffer library (explained in Installation section).
- Set CONFIG_RTE_LIBRTE_PMD_AESNI_GCM=y in config/common_base.

To use the PMD in an application, user must:

- Call rte_vdev_init("crypto_aesni_gcm") within the application.
- Use –vdev="crypto_aesni_gcm" in the EAL options, which will call rte_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 1 -n 4 --vdev="crypto_aesni_gcm,socket_id=0,max_nb_sessions=128" \
-- -p 1 --cdev SW --chain AEAD --aead_algo "aes-gcm"
```

11.4 ARMv8 Crypto Poll Mode Driver

This code provides the initial implementation of the ARMv8 crypto PMD. The driver uses ARMv8 cryptographic extensions to process chained crypto operations in an optimized way. The core functionality is provided by a low-level library, written in the assembly code.

11.4.1 Features

ARMv8 Crypto PMD has support for the following algorithm pairs:

Supported cipher algorithms:

- `RTE_CRYPTO_CIPHER_AES_CBC`

Supported authentication algorithms:

- `RTE_CRYPTO_AUTH_SHA1_HMAC`

- RTE_CRYPTO_AUTH_SHA256_HMAC

11.4.2 Installation

In order to enable this virtual crypto PMD, user must:

- Download AArch64 crypto library source code from here
- Export the environmental variable ARMV8_CRYPTO_LIB_PATH with the path to AArch64cryptolib library.
- Build the library by invoking:

```
make -C $ARMV8_CRYPTO_LIB_PATH/
```

- Set CONFIG_RTE_LIBRTE_PMD_ARMV8_CRYPTO=y in config/defconfig_arm64-armv8a-linux-gcc

The corresponding device can be created only if the following features are supported by the CPU:

- RTE_CPUFLAG_AES
- RTE_CPUFLAG_SHA1
- RTE_CPUFLAG_SHA2
- RTE_CPUFLAG_NEON

11.4.3 Initialization

User can use app/test application to check how to use this PMD and to verify crypto processing.

Test name is cryptodev_sw_armv8_autotest.

11.4.4 Limitations

- Maximum number of sessions is 2048.
- Only chained operations are supported.
- AES-128-CBC is the only supported cipher variant.
- Cipher input data has to be a multiple of 16 bytes.
- Digest input data has to be a multiple of 8 bytes.

11.5 NXP CAAM JOB RING (caam_jr)

The caam_jr PMD provides poll mode crypto driver support for NXP SEC 4.x+ (CAAM) hardware accelerator. More information is available at:

NXP Cryptographic Acceleration Technology.

11.5.1 Architecture

SEC is the SOC's security engine, which serves as NXP's latest cryptographic acceleration and offloading hardware. It combines functions previously implemented in separate modules to create a modular and scalable acceleration and assurance engine. It also implements block encryption algorithms, stream cipher algorithms, hashing algorithms, public key algorithms, run-time integrity checking, and a hardware random number generator. SEC performs higher-level cryptographic operations than previous NXP cryptographic accelerators. This provides significant improvement to system level performance.

SEC HW accelerator above 4.x+ version are also known as CAAM.

caam_jr PMD is one of DPAA drivers which uses uio interface to interact with Linux kernel for configure and destroy the device instance (ring).

11.5.2 Implementation

SEC provides platform assurance by working with SecMon, which is a companion logic block that tracks the security state of the SOC. SEC is programmed by means of descriptors (not to be confused with frame descriptors (FDs)) that indicate the operations to be performed and link to the message and associated data. SEC incorporates two DMA engines to fetch the descriptors, read the message data, and write the results of the operations. The DMA engine provides a scatter/gather capability so that SEC can read and write data scattered in memory. SEC may be configured by means of software for dynamic changes in byte ordering. The default configuration for this version of SEC is little-endian mode.

Note that one physical Job Ring represent one caam_jr device.

11.5.3 Features

The CAAM_JR PMD has support for:

Cipher algorithms:

- `RTE_CRYPTO_CIPHER_3DES_CBC`
- `RTE_CRYPTO_CIPHER_AES128_CBC`
- `RTE_CRYPTO_CIPHER_AES192_CBC`
- `RTE_CRYPTO_CIPHER_AES256_CBC`
- `RTE_CRYPTO_CIPHER_AES128_CTR`
- `RTE_CRYPTO_CIPHER_AES192_CTR`
- `RTE_CRYPTO_CIPHER_AES256_CTR`

Hash algorithms:

- `RTE_CRYPTO_AUTH_SHA1_HMAC`
- `RTE_CRYPTO_AUTH_SHA224_HMAC`
- `RTE_CRYPTO_AUTH_SHA256_HMAC`
- `RTE_CRYPTO_AUTH_SHA384_HMAC`
- `RTE_CRYPTO_AUTH_SHA512_HMAC`
- `RTE_CRYPTO_AUTH_MD5_HMAC`

AEAD algorithms:

- RTE_CRYPTO_AEAD_AES_GCM

11.5.4 Supported DPAA SoCs

- LS1046A/LS1026A
- LS1043A/LS1023A
- LS1028A
- LS1012A

11.5.5 Limitations

- Hash followed by Cipher mode is not supported
- Only supports the session-oriented API implementation (session-less APIs are not supported).

11.5.6 Prerequisites

caam_jr driver has following dependencies are not part of DPDK and must be installed separately:

- **NXP Linux SDK**

  NXP Linux software development kit (SDK) includes support for the family of QorIQ® ARM-Architecture-based system on chip (SoC) processors and corresponding boards.

  It includes the Linux board support packages (BSPs) for NXP SoCs, a fully operational tool chain, kernel and board specific modules.

  SDK and related information can be obtained from: NXP QorIQ SDK.

Currently supported by DPDK:

- NXP SDK **18.09+**.
- Supported architectures: **arm64 LE**.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

11.5.7 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file to enable caam_jr PMD.

Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_PMD_CAAM_JR` (default `n`) By default it is only enabled in common_linux config. Toggle compilation of the `librte_pmd_caam_jr` driver.
- `CONFIG_RTE_LIBRTE_PMD_CAAM_JR_BE` (default `n`) By default it is disabled. It can be used when the underlying hardware supports the CAAM in BE mode. LS1043A, LS1046A and LS1012A support CAAM in BE mode. LS1028A supports CAAM in LE mode.

11.5.8 Installations

To compile the caam_jr PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-armv8a-linux-gcc install
```

11.5.9 Enabling logs

For enabling logs, use the following EAL parameter:

```
./your_crypto_application <EAL args> --log-level=pmd.crypto.caam,<level>
```

11.6 AMD CCP Poll Mode Driver

This code provides the initial implementation of the ccp poll mode driver. The CCP poll mode driver library (librte_pmd_ccp) implements support for AMD's cryptographic co-processor (CCP). The CCP PMD is a virtual crypto poll mode driver which schedules crypto operations to one or more available CCP hardware engines on the platform. The CCP PMD provides poll mode crypto driver support for the following hardware accelerator devices:

```
AMD Cryptographic Co-processor (0x1456)
AMD Cryptographic Co-processor (0x1468)
```

11.6.1 Features

CCP crypto PMD has support for:

Cipher algorithms:

- RTE_CRYPTO_CIPHER_AES_CBC
- RTE_CRYPTO_CIPHER_AES_ECB
- RTE_CRYPTO_CIPHER_AES_CTR
- RTE_CRYPTO_CIPHER_3DES_CBC

Hash algorithms:

- RTE_CRYPTO_AUTH_SHA1
- RTE_CRYPTO_AUTH_SHA1_HMAC
- RTE_CRYPTO_AUTH_SHA224
- RTE_CRYPTO_AUTH_SHA224_HMAC
- RTE_CRYPTO_AUTH_SHA256
- RTE_CRYPTO_AUTH_SHA256_HMAC
- RTE_CRYPTO_AUTH_SHA384
- RTE_CRYPTO_AUTH_SHA384_HMAC

- RTE_CRYPTO_AUTH_SHA512
- RTE_CRYPTO_AUTH_SHA512_HMAC
- RTE_CRYPTO_AUTH_MD5_HMAC
- RTE_CRYPTO_AUTH_AES_CMAC
- RTE_CRYPTO_AUTH_SHA3_224
- RTE_CRYPTO_AUTH_SHA3_224_HMAC
- RTE_CRYPTO_AUTH_SHA3_256
- RTE_CRYPTO_AUTH_SHA3_256_HMAC
- RTE_CRYPTO_AUTH_SHA3_384
- RTE_CRYPTO_AUTH_SHA3_384_HMAC
- RTE_CRYPTO_AUTH_SHA3_512
- RTE_CRYPTO_AUTH_SHA3_512_HMAC

AEAD algorithms:

- RTE_CRYPTO_AEAD_AES_GCM

11.6.2 Installation

To compile ccp PMD, it has to be enabled in the config/common_base file and openssl packages have to be installed in the build environment.

- CONFIG_RTE_LIBRTE_PMD_CCP=y

For Ubuntu 16.04 LTS use below to install openssl in the build system:

```
sudo apt-get install openssl
```

This code was verified on Ubuntu 16.04.

11.6.3 Initialization

Bind the CCP devices to DPDK UIO driver module before running the CCP PMD stack. e.g. for the 0x1456 device:

```
cd to the top-level DPDK directory
modprobe uio
insmod ./build/kmod/igb_uio.ko
echo "1022 1456" > /sys/bus/pci/drivers/igb_uio/new_id
```

Another way to bind the CCP devices to DPDK UIO driver is by using the `dpdk-devbind.py` script. The following command assumes `BFD` as `0000:09:00.2`:

```
cd to the top-level DPDK directory
./usertools/dpdk-devbind.py -b igb_uio 0000:09:00.2
```

In order to enable the ccp crypto PMD, user must set CONFIG_RTE_LIBRTE_PMD_CCP=y in config/common_base.

To use the PMD in an application, user must:

- Call rte_vdev_init("crypto_ccp") within the application.
- Use –vdev="crypto_ccp" in the EAL options, which will call rte_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated. (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device.
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).
- ccp_auth_opt: Specify authentication operations to perform on CPU using openssl APIs.

To validate ccp pmd, l2fwd-crypto example can be used with following command:

```
sudo ./build/l2fwd-crypto -l 1 -n 4 --vdev "crypto_ccp" -- -p 0x1
--chain CIPHER_HASH --cipher_op ENCRYPT --cipher_algo aes-cbc
--cipher_key 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f
--cipher_iv 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:ff
--auth_op GENERATE --auth_algo sha1-hmac
--auth_key 11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
```

The CCP PMD also supports computing authentication over CPU with cipher offloaded to CCP. To enable this feature, pass an additional argument as ccp_auth_opt=1 to –vdev parameters as following:

```
sudo ./build/l2fwd-crypto -l 1 -n 4 --vdev "crypto_ccp,ccp_auth_opt=1" -- -p 0x1
--chain CIPHER_HASH --cipher_op ENCRYPT --cipher_algo aes-cbc
--cipher_key 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f
--cipher_iv 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:ff
--auth_op GENERATE --auth_algo sha1-hmac
--auth_key 11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
```

11.6.4 Limitations

- Chained mbufs are not supported.
- MD5_HMAC is supported only for CPU based authentication.

11.7 NXP DPAA2 CAAM (DPAA2_SEC)

The DPAA2_SEC PMD provides poll mode crypto driver support for NXP DPAA2 CAAM hardware accelerator.

11.7.1 Architecture

SEC is the SOC's security engine, which serves as NXP's latest cryptographic acceleration and offloading hardware. It combines functions previously implemented in separate modules to create a modular and scalable acceleration and assurance engine. It also implements block encryption algorithms, stream cipher algorithms, hashing algorithms, public key algorithms, run-time integrity checking, and a hardware random number generator. SEC performs higher-level cryptographic operations than previous NXP cryptographic accelerators. This provides significant improvement to system level performance.

DPAA2_SEC is one of the hardware resource in DPAA2 Architecture. More information on DPAA2 Architecture is described in *DPAA2 Overview*.

DPAA2_SEC PMD is one of DPAA2 drivers which interacts with Management Complex (MC) portal to access the hardware object - DPSECI. The MC provides access to create, discover, connect, configure and destroy dpseci objects in DPAA2_SEC PMD.

DPAA2_SEC PMD also uses some of the other hardware resources like buffer pools, queues, queue portals to store and to enqueue/dequeue data to the hardware SEC.

DPSECI objects are detected by PMD using a resource container called DPRC (like in *DPAA2 Overview*).

For example:

```
DPRC.1 (bus)
  |
  +--+--------+-------+-------+-------+---------+
     |        |       |       |       |         |
   DPMCP.1  DPIO.1  DPBP.1  DPNI.1  DPMAC.1  DPSECI.1
   DPMCP.2  DPIO.2          DPNI.2  DPMAC.2  DPSECI.2
   DPMCP.3
```

11.7.2 Implementation

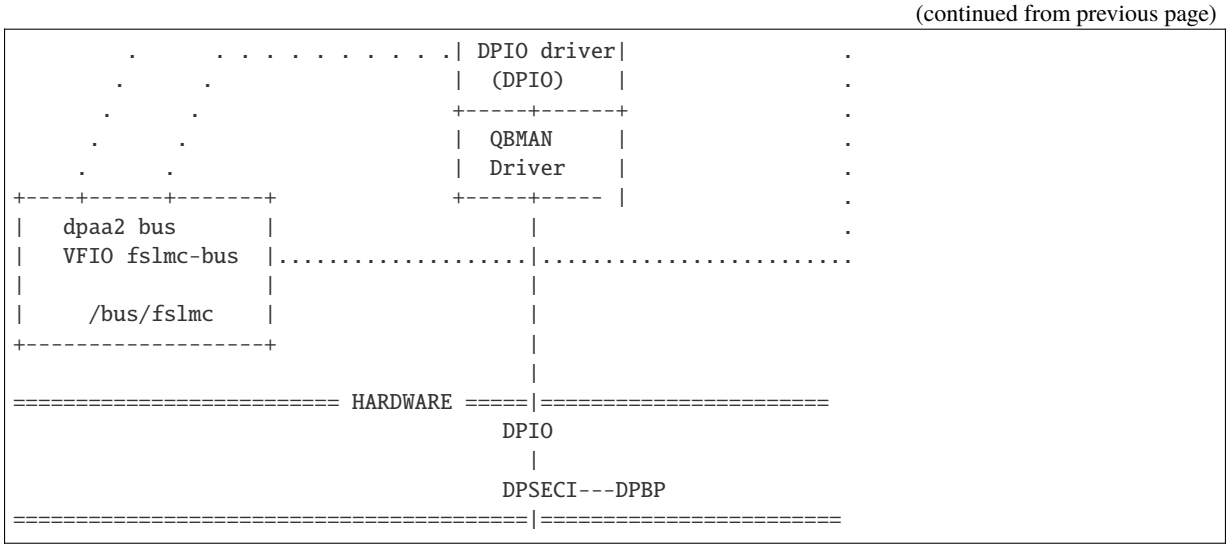
SEC provides platform assurance by working with SecMon, which is a companion logic block that tracks the security state of the SOC. SEC is programmed by means of descriptors (not to be confused with frame descriptors (FDs)) that indicate the operations to be performed and link to the message and associated data. SEC incorporates two DMA engines to fetch the descriptors, read the message data, and write the results of the operations. The DMA engine provides a scatter/gather capability so that SEC can read and write data scattered in memory. SEC may be configured by means of software for dynamic changes in byte ordering. The default configuration for this version of SEC is little-endian mode.

A block diagram similar to dpaa2 NIC is shown below to show where DPAA2_SEC fits in the DPAA2 Bus model

```
                              +-----------------+
                              | DPDK DPAA2_SEC  |
                              |     PMD         |
                              +-----------------+       +-------------+
                              |  MC SEC object  |.......|  Mempool    |
              . . . . . . . . |   (DPSECI)      |       |  (DPBP)     |
            .                 +---+---+---------+       +-----+------+
           .                      ^   |                       .
          .                       |   |<enqueue,              .
         .                        |   | dequeue>              .
        .                         |   |                       .
       .                          +---+---V-----+               .
```

(continues on next page)

(continued from previous page)



11.7.3 Features

The DPAA2_SEC PMD has support for:

Cipher algorithms:

- RTE_CRYPTO_CIPHER_3DES_CBC
- RTE_CRYPTO_CIPHER_AES128_CBC
- RTE_CRYPTO_CIPHER_AES192_CBC
- RTE_CRYPTO_CIPHER_AES256_CBC
- RTE_CRYPTO_CIPHER_AES128_CTR
- RTE_CRYPTO_CIPHER_AES192_CTR
- RTE_CRYPTO_CIPHER_AES256_CTR

Hash algorithms:

- RTE_CRYPTO_AUTH_SHA1_HMAC
- RTE_CRYPTO_AUTH_SHA224_HMAC
- RTE_CRYPTO_AUTH_SHA256_HMAC
- RTE_CRYPTO_AUTH_SHA384_HMAC
- RTE_CRYPTO_AUTH_SHA512_HMAC
- RTE_CRYPTO_AUTH_MD5_HMAC

AEAD algorithms:

- RTE_CRYPTO_AEAD_AES_GCM

11.7.4 Supported DPAA2 SoCs

- LS2160A
- LS2084A/LS2044A
- LS2088A/LS2048A
- LS1088A/LS1048A

11.7.5 Whitelisting & Blacklisting

For blacklisting a DPAA2 SEC device, following commands can be used.

```
<dpdk app> <EAL args> -b "fslmc:dpseci.x" -- ...
```

Where x is the device object id as configured in resource container.

11.7.6 Limitations

- Hash followed by Cipher mode is not supported
- Only supports the session-oriented API implementation (session-less APIs are not supported).

11.7.7 Prerequisites

DPAA2_SEC driver has similar pre-requisites as described in *DPAA2 Overview*. The following dependencies are not part of DPDK and must be installed separately:

See *NXP QorIQ DPAA2 Board Support Package* for setup information

Currently supported by DPDK:

- NXP SDK **19.09+**.
- MC Firmware version **10.18.0** and higher.
- Supported architectures: **arm64 LE**.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

11.7.8 Pre-Installation Configuration

Config File Options

Basic DPAA2 config file options are described in *DPAA2 Overview*. In addition to those, the following options can be modified in the `config` file to enable DPAA2_SEC PMD.

Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_PMD_DPAA2_SEC` (default `n`) By default it is only enabled in defconfig_arm64-dpaa-* config. Toggle compilation of the `librte_pmd_dpaa2_sec` driver.

11.7.9 Installations

To compile the DPAA2_SEC PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-dpaa-linux-gcc install
```

11.7.10 Enabling logs

For enabling logs, use the following EAL parameter:

```
./your_crypto_application <EAL args> --log-level=pmd.crypto.dpaa2:<level>
```

Using `crypto.dpaa2` as log matching criteria, all Crypto PMD logs can be enabled which are lower than logging `level`.

11.8 NXP DPAA CAAM (DPAA_SEC)

The DPAA_SEC PMD provides poll mode crypto driver support for NXP DPAA CAAM hardware accelerator.

11.8.1 Architecture

SEC is the SOC's security engine, which serves as NXP's latest cryptographic acceleration and offloading hardware. It combines functions previously implemented in separate modules to create a modular and scalable acceleration and assurance engine. It also implements block encryption algorithms, stream cipher algorithms, hashing algorithms, public key algorithms, run-time integrity checking, and a hardware random number generator. SEC performs higher-level cryptographic operations than previous NXP cryptographic accelerators. This provides significant improvement to system level performance.

DPAA_SEC is one of the hardware resource in DPAA Architecture. More information on DPAA Architecture is described in *DPAA Overview*.

DPAA_SEC PMD is one of DPAA drivers which interacts with QBMAN to create, configure and destroy the device instance using queue pair with CAAM portal.

DPAA_SEC PMD also uses some of the other hardware resources like buffer pools, queues, queue portals to store and to enqueue/dequeue data to the hardware SEC.

11.8.2 Implementation

SEC provides platform assurance by working with SecMon, which is a companion logic block that tracks the security state of the SOC. SEC is programmed by means of descriptors (not to be confused with frame descriptors (FDs)) that indicate the operations to be performed and link to the message and associated data. SEC incorporates two DMA engines to fetch the descriptors, read the message data, and write the results of the operations. The DMA engine provides a scatter/gather capability so that SEC can read and write data scattered in memory. SEC may be configured by means of software for dynamic changes in byte ordering. The default configuration for this version of SEC is little-endian mode.

11.8.3 Features

The DPAA PMD has support for:

Cipher algorithms:

- RTE_CRYPTO_CIPHER_3DES_CBC
- RTE_CRYPTO_CIPHER_AES128_CBC
- RTE_CRYPTO_CIPHER_AES192_CBC
- RTE_CRYPTO_CIPHER_AES256_CBC
- RTE_CRYPTO_CIPHER_AES128_CTR
- RTE_CRYPTO_CIPHER_AES192_CTR
- RTE_CRYPTO_CIPHER_AES256_CTR
- RTE_CRYPTO_CIPHER_SNOW3G_UEA2
- RTE_CRYPTO_CIPHER_ZUC_EEA3

Hash algorithms:

- RTE_CRYPTO_AUTH_SHA1_HMAC
- RTE_CRYPTO_AUTH_SHA224_HMAC
- RTE_CRYPTO_AUTH_SHA256_HMAC
- RTE_CRYPTO_AUTH_SHA384_HMAC
- RTE_CRYPTO_AUTH_SHA512_HMAC
- RTE_CRYPTO_AUTH_SNOW3G_UIA2
- RTE_CRYPTO_AUTH_MD5_HMAC
- RTE_CRYPTO_AUTH_ZUC_EIA3

AEAD algorithms:

- RTE_CRYPTO_AEAD_AES_GCM

11.8.4 Supported DPAA SoCs

- LS1046A/LS1026A
- LS1043A/LS1023A

11.8.5 Whitelisting & Blacklisting

For blacklisting a DPAA device, following commands can be used.

```
<dpdk app> <EAL args> -b "dpaa:dpaa_sec-X" -- ...
e.g. "dpaa:dpaa_sec-1"

or to disable all 4 SEC devices
-b "dpaa:dpaa_sec-1"  -b "dpaa:dpaa_sec-2" -b "dpaa:dpaa_sec-3" -b "dpaa:dpaa_sec-4"
```

11.8.6 Limitations

- Hash followed by Cipher mode is not supported
- Only supports the session-oriented API implementation (session-less APIs are not supported).

11.8.7 Prerequisites

DPAA_SEC driver has similar pre-requisites as described in *DPAA Overview*.

See *NXP QorIQ DPAA Board Support Package* for setup information

- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

11.8.8 Pre-Installation Configuration

Config File Options

Basic DPAA config file options are described in *DPAA Overview*. In addition to those, the following options can be modified in the `config` file to enable DPAA_SEC PMD.

Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_PMD_DPAA_SEC` (default `n`) By default it is only enabled in defconfig_arm64-dpaa-* config. Toggle compilation of the `librte_pmd_dpaa_sec` driver.

11.8.9 Installations

To compile the DPAA_SEC PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-dpaa-linux-gcc install
```

11.8.10 Enabling logs

For enabling logs, use the following EAL parameter:

```
./your_crypto_application <EAL args> --log-level=pmd.crypto.dpaa:<level>
```

Using `pmd.crypto.dpaa` as log matching criteria, all Crypto PMD logs can be enabled which are lower than logging `level`.

11.9 KASUMI Crypto Poll Mode Driver

The KASUMI PMD (**librte_pmd_kasumi**) provides poll mode crypto driver support for utilizing Intel IPSec Multi-buffer library which implements F8 and F9 functions for KASUMI UEA1 cipher and UIA1 hash algorithms.

11.9.1 Features

KASUMI PMD has support for:

Cipher algorithm:

- RTE_CRYPTO_CIPHER_KASUMI_F8

Authentication algorithm:

- RTE_CRYPTO_AUTH_KASUMI_F9

11.9.2 Limitations

- Chained mbufs are not supported.
- KASUMI(F9) supported only if hash offset and length field is byte-aligned.
- In-place bit-level operations for KASUMI(F8) are not supported (if length and/or offset of data to be ciphered is not byte-aligned).

11.9.3 Installation

To build DPDK with the KASUMI_PMD the user is required to download the multi-buffer library from here and compile it on their user system before building DPDK. The latest version of the library supported by this PMD is v0.54, which can be downloaded from https://github.com/01org/intel-ipsec-mb/archive/v0.54.zip.

After downloading the library, the user needs to unpack and compile it on their system before building DPDK:

```
make
make install
```

The library requires NASM to be built. Depending on the library version, it might require a minimum NASM version (e.g. v0.54 requires at least NASM 2.14).

NASM is packaged for different OS. However, on some OS the version is too old, so a manual installation is required. In that case, NASM can be downloaded from NASM website. Once it is downloaded, extract it and follow these steps:

```
./configure
make
make install
```

Note: Compilation of the Multi-Buffer library is broken when GCC < 5.0, if library <= v0.53. If a lower GCC version than 5.0, the workaround proposed by the following link should be used: https://github.com/intel/intel-ipsec-mb/issues/40.

As a reference, the following table shows a mapping between the past DPDK versions and the external crypto libraries supported by them:

Table 11.8: DPDK and external crypto library version compatibility

DPDK version	Crypto library version
16.11 - 19.11	LibSSO KASUMI
20.02+	Multi-buffer library 0.53 - 0.54

11.9.4 Initialization

In order to enable this virtual crypto PMD, user must:

- Build the multi buffer library (explained in Installation section).
- Build DPDK as follows:

```
make config T=x86_64-native-linux-gcc
sed -i 's,\(CONFIG_RTE_LIBRTE_PMD_KASUMI\)=n,\1=y,' build/.config
make
```

To use the PMD in an application, user must:

- Call rte_vdev_init("crypto_kasumi") within the application.
- Use –vdev="crypto_kasumi" in the EAL options, which will call rte_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 1 -n 4 --vdev="crypto_kasumi,socket_id=0,max_nb_sessions=128" \
-- -p 1 --cdev SW --chain CIPHER_ONLY --cipher_algo "kasumi-f8"
```

11.9.5 Extra notes on KASUMI F9

When using KASUMI F9 authentication algorithm, the input buffer must be constructed according to the 3GPP KASUMI specifications (section 4.4, page 13): http://cryptome.org/3gpp/35201-900.pdf. Input buffer has to have COUNT (4 bytes), FRESH (4 bytes), MESSAGE and DIRECTION (1 bit) concatenated. After the DIRECTION bit, a single '1' bit is appended, followed by between 0 and 7 '0' bits, so that the total length of the buffer is multiple of 8 bits. Note that the actual message can be any length, specified in bits.

Once this buffer is passed this way, when creating the crypto operation, length of data to authenticate (op.sym.auth.data.length) must be the length of all the items described above, including the padding at the end. Also, offset of data to authenticate (op.sym.auth.data.offset) must be such that points at the start of the COUNT bytes.

11.10 Cavium OCTEON TX Crypto Poll Mode Driver

The OCTEON TX crypto poll mode driver provides support for offloading cryptographic operations to cryptographic accelerator units on **OCTEON TX** ® family of processors (CN8XXX). The OCTEON TX crypto poll mode driver enqueues the crypto request to this accelerator and dequeues the response once the operation is completed.

11.10.1 Supported Symmetric Crypto Algorithms

Cipher Algorithms

- RTE_CRYPTO_CIPHER_NULL
- RTE_CRYPTO_CIPHER_3DES_CBC
- RTE_CRYPTO_CIPHER_3DES_ECB
- RTE_CRYPTO_CIPHER_AES_CBC
- RTE_CRYPTO_CIPHER_AES_CTR
- RTE_CRYPTO_CIPHER_AES_XTS
- RTE_CRYPTO_CIPHER_DES_CBC
- RTE_CRYPTO_CIPHER_KASUMI_F8
- RTE_CRYPTO_CIPHER_SNOW3G_UEA2
- RTE_CRYPTO_CIPHER_ZUC_EEA3

Hash Algorithms

- RTE_CRYPTO_AUTH_NULL
- RTE_CRYPTO_AUTH_AES_GMAC
- RTE_CRYPTO_AUTH_KASUMI_F9
- RTE_CRYPTO_AUTH_MD5
- RTE_CRYPTO_AUTH_MD5_HMAC
- RTE_CRYPTO_AUTH_SHA1
- RTE_CRYPTO_AUTH_SHA1_HMAC
- RTE_CRYPTO_AUTH_SHA224
- RTE_CRYPTO_AUTH_SHA224_HMAC
- RTE_CRYPTO_AUTH_SHA256
- RTE_CRYPTO_AUTH_SHA256_HMAC
- RTE_CRYPTO_AUTH_SHA384
- RTE_CRYPTO_AUTH_SHA384_HMAC
- RTE_CRYPTO_AUTH_SHA512
- RTE_CRYPTO_AUTH_SHA512_HMAC
- RTE_CRYPTO_AUTH_SNOW3G_UIA2
- RTE_CRYPTO_AUTH_ZUC_EIA3

AEAD Algorithms

- RTE_CRYPTO_AEAD_AES_GCM

11.10.2 Supported Asymmetric Crypto Algorithms

- RTE_CRYPTO_ASYM_XFORM_RSA
- RTE_CRYPTO_ASYM_XFORM_MODEX

11.10.3 Config flags

For compiling the OCTEON TX crypto poll mode driver, please check if the CONFIG_RTE_LIBRTE_PMD_OCTEONTX_CRYPTO setting is set to *y* in config/common_base file.

- CONFIG_RTE_LIBRTE_PMD_OCTEONTX_CRYPTO=y

11.10.4 Compilation

The OCTEON TX crypto poll mode driver can be compiled either natively on **OCTEON TX** ® board or cross-compiled on an x86 based platform.

Refer *OCTEON TX Board Support Package* for details about setting up the platform and building DPDK applications.

Note: OCTEON TX crypto PF driver needs microcode to be available at */lib/firmware/* directory. Refer SDK documents for further information.

SDK and related information can be obtained from: Cavium support site.

11.10.5 Execution

The number of crypto VFs to be enabled can be controlled by setting sysfs entry, *sriov_numvfs*, for the corresponding PF driver.

```
echo <num_vfs> > /sys/bus/pci/devices/<dev_bus_id>/sriov_numvfs
```

The device bus ID, *dev_bus_id*, to be used in the above step can be found out by using dpdk-devbind.py script. The OCTEON TX crypto PF device need to be identified and the corresponding device number can be used to tune various PF properties.

Once the required VFs are enabled, dpdk-devbind.py script can be used to identify the VFs. To be accessible from DPDK, VFs need to be bound to vfio-pci driver:

```
cd <dpdk directory>
./usertools/dpdk-devbind.py -u <vf device no>
./usertools/dpdk-devbind.py -b vfio-pci <vf device no>
```

Appropriate huge page need to be setup in order to run the DPDK example applications.

```
echo 8 > /sys/kernel/mm/hugepages/hugepages-524288kB/nr_hugepages
mkdir /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
```

Example applications can now be executed with crypto operations offloaded to OCTEON TX crypto PMD.

```
./build/ipsec-secgw --log-level=8 -c 0xff -- -P -p 0x3 -u 0x2 --config
"(1,0,0),(0,0,0)" -f ep1.cfg
```

11.10.6 Testing

The symmetric crypto operations on OCTEON TX crypto PMD may be verified by running the test application:

```
./test
RTE>>cryptodev_octeontx_autotest
```

The asymmetric crypto operations on OCTEON TX crypto PMD may be verified by running the test application:

```
./test
RTE>>cryptodev_octeontx_asym_autotest
```

11.11 Marvell OCTEON TX2 Crypto Poll Mode Driver

The OCTEON TX2 crypto poll mode driver provides support for offloading cryptographic operations to cryptographic accelerator units on the **OCTEON TX2** ® family of processors (CN9XXX).

More information about OCTEON TX2 SoCs may be obtained from https://www.marvell.com

11.11.1 Features

The OCTEON TX2 crypto PMD has support for:

Symmetric Crypto Algorithms

Cipher algorithms:

- RTE_CRYPTO_CIPHER_NULL
- RTE_CRYPTO_CIPHER_3DES_CBC
- RTE_CRYPTO_CIPHER_3DES_ECB
- RTE_CRYPTO_CIPHER_AES_CBC
- RTE_CRYPTO_CIPHER_AES_CTR
- RTE_CRYPTO_CIPHER_AES_XTS
- RTE_CRYPTO_CIPHER_DES_CBC
- RTE_CRYPTO_CIPHER_KASUMI_F8
- RTE_CRYPTO_CIPHER_SNOW3G_UEA2
- RTE_CRYPTO_CIPHER_ZUC_EEA3

Hash algorithms:

- RTE_CRYPTO_AUTH_NULL
- RTE_CRYPTO_AUTH_AES_GMAC
- RTE_CRYPTO_AUTH_KASUMI_F9
- RTE_CRYPTO_AUTH_MD5
- RTE_CRYPTO_AUTH_MD5_HMAC
- RTE_CRYPTO_AUTH_SHA1
- RTE_CRYPTO_AUTH_SHA1_HMAC
- RTE_CRYPTO_AUTH_SHA224
- RTE_CRYPTO_AUTH_SHA224_HMAC
- RTE_CRYPTO_AUTH_SHA256

- RTE_CRYPTO_AUTH_SHA256_HMAC
- RTE_CRYPTO_AUTH_SHA384
- RTE_CRYPTO_AUTH_SHA384_HMAC
- RTE_CRYPTO_AUTH_SHA512
- RTE_CRYPTO_AUTH_SHA512_HMAC
- RTE_CRYPTO_AUTH_SNOW3G_UIA2
- RTE_CRYPTO_AUTH_ZUC_EIA3

AEAD algorithms:

- RTE_CRYPTO_AEAD_AES_GCM

Asymmetric Crypto Algorithms

- RTE_CRYPTO_ASYM_XFORM_RSA
- RTE_CRYPTO_ASYM_XFORM_MODEX

11.11.2 Installation

The OCTEON TX2 crypto PMD may be compiled natively on an OCTEON TX2 platform or cross-compiled on an x86 platform.

Enable OCTEON TX2 crypto PMD in your config file:

- CONFIG_RTE_LIBRTE_PMD_OCTEONTX2_CRYPTO=y

Refer to *Marvell OCTEON TX2 Platform Guide* for instructions to build your DPDK application.

Note: The OCTEON TX2 crypto PMD uses services from the kernel mode OCTEON TX2 crypto PF driver in linux. This driver is included in the OCTEON TX SDK.

11.11.3 Initialization

List the CPT PF devices available on your OCTEON TX2 platform:

```
lspci -d:a0fd
```

`a0fd` is the CPT PF device id. You should see output similar to:

```
0002:10:00.0 Class 1080: Device 177d:a0fd
```

Set `sriov_numvfs` on the CPT PF device, to create a VF:

```
echo 1 > /sys/bus/pci/drivers/octeontx2-cpt/0002:10:00.0/sriov_numvfs
```

Bind the CPT VF device to the vfio_pci driver:

```
echo '177d a0fe' > /sys/bus/pci/drivers/vfio-pci/new_id
echo 0002:10:00.1 > /sys/bus/pci/devices/0002:10:00.1/driver/unbind
echo 0002:10:00.1 > /sys/bus/pci/drivers/vfio-pci/bind
```

Another way to bind the VF would be to use the `dpdk-devbind.py` script:

```
cd <dpdk directory>
./usertools/dpdk-devbind.py -u 0002:10:00.1
./usertools/dpdk-devbind.py -b vfio-pci 0002:10.00.1
```

Note: Ensure that sufficient huge pages are available for your application:

```
echo 8 > /sys/kernel/mm/hugepages/hugepages-524288kB/nr_hugepages
```

Refer to *Use of Hugepages in the Linux Environment* for more details.

11.11.4 Debugging Options

Table 11.9: OCTEON TX2 crypto PMD debug options

#	Component	EAL log command
1	CPT	–log-level='pmd.crypto.octeontx2,8'

11.11.5 Testing

The symmetric crypto operations on OCTEON TX2 crypto PMD may be verified by running the test application:

```
./test
RTE>>cryptodev_octeontx2_autotest
```

The asymmetric crypto operations on OCTEON TX2 crypto PMD may be verified by running the test application:

```
./test
RTE>>cryptodev_octeontx2_asym_autotest
```

11.12 OpenSSL Crypto Poll Mode Driver

This code provides the initial implementation of the openssl poll mode driver. All cryptography operations are using Openssl library crypto API. Each algorithm uses EVP interface from openssl API - which is recommended by Openssl maintainers.

For more details about openssl library please visit openssl webpage: https://www.openssl.org/

11.12.1 Features

OpenSSL PMD has support for:

Supported cipher algorithms:

- RTE_CRYPTO_CIPHER_3DES_CBC
- RTE_CRYPTO_CIPHER_AES_CBC
- RTE_CRYPTO_CIPHER_AES_CTR
- RTE_CRYPTO_CIPHER_3DES_CTR
- RTE_CRYPTO_CIPHER_DES_DOCSISBPI

Supported authentication algorithms:

- RTE_CRYPTO_AUTH_AES_GMAC
- RTE_CRYPTO_AUTH_MD5
- RTE_CRYPTO_AUTH_SHA1
- RTE_CRYPTO_AUTH_SHA224
- RTE_CRYPTO_AUTH_SHA256
- RTE_CRYPTO_AUTH_SHA384
- RTE_CRYPTO_AUTH_SHA512
- RTE_CRYPTO_AUTH_MD5_HMAC
- RTE_CRYPTO_AUTH_SHA1_HMAC
- RTE_CRYPTO_AUTH_SHA224_HMAC
- RTE_CRYPTO_AUTH_SHA256_HMAC
- RTE_CRYPTO_AUTH_SHA384_HMAC
- RTE_CRYPTO_AUTH_SHA512_HMAC

Supported AEAD algorithms:

- RTE_CRYPTO_AEAD_AES_GCM
- RTE_CRYPTO_AEAD_AES_CCM

Supported Asymmetric Crypto algorithms:

- RTE_CRYPTO_ASYM_XFORM_RSA
- RTE_CRYPTO_ASYM_XFORM_DSA
- RTE_CRYPTO_ASYM_XFORM_DH
- RTE_CRYPTO_ASYM_XFORM_MODINV
- RTE_CRYPTO_ASYM_XFORM_MODEX

11.12.2 Installation

To compile openssl PMD, it has to be enabled in the config/common_base file and appropriate openssl packages have to be installed in the build environment.

The newest openssl library version is supported:

- 1.0.2h-fips 3 May 2016.

Older versions that were also verified:

- 1.0.1f 6 Jan 2014
- 1.0.1 14 Mar 2012

For Ubuntu 14.04 LTS these packages have to be installed in the build system:

```
sudo apt-get install openssl
sudo apt-get install libc6-dev-i386 # for i686-native-linux-gcc target
```

This code was also verified on Fedora 24. This code has NOT been verified on FreeBSD yet.

11.12.3 Initialization

User can use app/test application to check how to use this pmd and to verify crypto processing.

Test name is cryptodev_openssl_autotest. For asymmetric crypto operations testing, run cryptodev_openssl_asym_autotest.

To verify real traffic l2fwd-crypto example can be used with this command:

```
sudo ./build/l2fwd-crypto -l 0-1 -n 4 --vdev "crypto_openssl"
--vdev "crypto_openssl"-- -p 0x3 --chain CIPHER_HASH
--cipher_op ENCRYPT --cipher_algo AES_CBC
--cipher_key 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f
--iv 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:ff
--auth_op GENERATE --auth_algo SHA1_HMAC
--auth_key 11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
```

11.12.4 Limitations

- Maximum number of sessions is 2048.
- Chained mbufs are supported only for source mbuf (destination must be contiguous).
- Hash only is not supported for GCM and GMAC.
- Cipher only is not supported for GCM and GMAC.

11.13 MVSAM Crypto Poll Mode Driver

The MVSAM CRYPTO PMD (**librte_crypto_mvsam_pmd**) provides poll mode crypto driver support by utilizing MUSDK library, which provides cryptographic operations acceleration by using Security Acceleration Engine (EIP197) directly from user-space with minimum overhead and high performance.

Detailed information about SoCs that use MVSAM crypto driver can be obtained here:

- https://www.marvell.com/embedded-processors/armada-70xx/
- https://www.marvell.com/embedded-processors/armada-80xx/
- https://www.marvell.com/embedded-processors/armada-3700/

11.13.1 Features

MVSAM CRYPTO PMD has support for:

Cipher algorithms:

- RTE_CRYPTO_CIPHER_NULL
- RTE_CRYPTO_CIPHER_AES_CBC
- RTE_CRYPTO_CIPHER_AES_CTR
- RTE_CRYPTO_CIPHER_AES_ECB
- RTE_CRYPTO_CIPHER_3DES_CBC
- RTE_CRYPTO_CIPHER_3DES_CTR
- RTE_CRYPTO_CIPHER_3DES_ECB

Hash algorithms:

- RTE_CRYPTO_AUTH_NULL
- RTE_CRYPTO_AUTH_MD5
- RTE_CRYPTO_AUTH_MD5_HMAC
- RTE_CRYPTO_AUTH_SHA1
- RTE_CRYPTO_AUTH_SHA1_HMAC
- RTE_CRYPTO_AUTH_SHA224
- RTE_CRYPTO_AUTH_SHA224_HMAC
- RTE_CRYPTO_AUTH_SHA256
- RTE_CRYPTO_AUTH_SHA256_HMAC
- RTE_CRYPTO_AUTH_SHA384
- RTE_CRYPTO_AUTH_SHA384_HMAC
- RTE_CRYPTO_AUTH_SHA512
- RTE_CRYPTO_AUTH_SHA512_HMAC
- RTE_CRYPTO_AUTH_AES_GMAC

AEAD algorithms:

- RTE_CRYPTO_AEAD_AES_GCM

For supported feature flags please consult *Crypto Device Supported Functionality Matrices*.

11.13.2 Limitations

- Hardware only supports scenarios where ICV (digest buffer) is placed just after the authenticated data. Other placement will result in error.

11.13.3 Installation

MVSAM CRYPTO PMD driver compilation is disabled by default due to external dependencies. Currently there are two driver specific compilation options in config/common_base available:

- CONFIG_RTE_LIBRTE_PMD_MVSAM_CRYPTO (default: n)

  Toggle compilation of the librte_pmd_mvsam driver.

MVSAM CRYPTO PMD requires MUSDK built with EIP197 support thus following extra option must be passed to the library configuration script:

```
--enable-sam [--enable-sam-statistics] [--enable-sam-debug]
```

For instructions how to build required kernel modules please refer to *doc/musdk_get_started.txt*.

11.13.4 Initialization

After successfully building MVSAM CRYPTO PMD, the following modules need to be loaded:

```
insmod musdk_cma.ko
insmod crypto_safexcel.ko rings=0,0
insmod mv_sam_uio.ko
```

The following parameters (all optional) are exported by the driver:

- max_nb_queue_pairs: maximum number of queue pairs in the device (default: 8 - A8K, 4 - A7K/A3K).
- max_nb_sessions: maximum number of sessions that can be created (default: 2048).
- socket_id: socket on which to allocate the device resources on.

l2fwd-crypto example application can be used to verify MVSAM CRYPTO PMD operation:

```
./l2fwd-crypto --vdev=eth_mvpp2,iface=eth0 --vdev=crypto_mvsam -- \
  --cipher_op ENCRYPT --cipher_algo aes-cbc \
  --cipher_key 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f  \
  --auth_op GENERATE --auth_algo sha1-hmac \
  --auth_key 10:11:12:13:14:15:16:17:18:19:1a:1b:1c:1d:1e:1f
```

11.14 Marvell NITROX Crypto Poll Mode Driver

The Nitrox crypto poll mode driver provides support for offloading cryptographic operations to the NITROX V security processor. Detailed information about the NITROX V security processor can be obtained here:

- https://www.marvell.com/security-solutions/nitrox-security-processors/nitrox-v/

11.14.1 Features

Nitrox crypto PMD has support for:

Cipher algorithms:

- `RTE_CRYPTO_CIPHER_AES_CBC`
- `RTE_CRYPTO_CIPHER_3DES_CBC`

Hash algorithms:

- `RTE_CRYPTO_AUTH_SHA1_HMAC`
- `RTE_CRYPTO_AUTH_SHA224_HMAC`
- `RTE_CRYPTO_AUTH_SHA256_HMAC`

11.14.2 Limitations

- AES_CBC Cipher Only combination is not supported.
- 3DES Cipher Only combination is not supported.
- Session-less APIs are not supported.

11.14.3 Installation

For compiling the Nitrox crypto PMD, please check if the CONFIG_RTE_LIBRTE_PMD_NITROX setting is set to *y* in config/common_base file.

- `CONFIG_RTE_LIBRTE_PMD_NITROX=y`

11.14.4 Initialization

Nitrox crypto PMD depend on Nitrox kernel PF driver being installed on the platform. Nitrox PF driver is required to create VF devices which will be used by the PMD. Each VF device can enable one cryptodev PMD.

Nitrox kernel PF driver is available as part of CNN55XX-Driver SDK. The SDK and it's installation instructions can be obtained from: Marvell Technical Documentation Portal.

11.15 Null Crypto Poll Mode Driver

The Null Crypto PMD (**librte_pmd_null_crypto**) provides a crypto poll mode driver which provides a minimal implementation for a software crypto device. As a null device it does not modify the data in the mbuf on which the crypto operation is to operate and it only has support for a single cipher and authentication algorithm.

When a burst of mbufs is submitted to a Null Crypto PMD for processing then each mbuf in the burst will be enqueued in an internal buffer for collection on a dequeue call as long as the mbuf has a valid rte_mbuf_offload operation with a valid rte_cryptodev_session or rte_crypto_xform chain of operations.

11.15.1 Features

Modes:

- RTE_CRYPTO_XFORM_CIPHER ONLY
- RTE_CRYPTO_XFORM_AUTH ONLY
- RTE_CRYPTO_XFORM_CIPHER THEN RTE_CRYPTO_XFORM_AUTH
- RTE_CRYPTO_XFORM_AUTH THEN RTE_CRYPTO_XFORM_CIPHER

Cipher algorithms:

- RTE_CRYPTO_CIPHER_NULL

Authentication algorithms:

- RTE_CRYPTO_AUTH_NULL

11.15.2 Limitations

- Only in-place is currently supported (destination address is the same as source address).

11.15.3 Installation

The Null Crypto PMD is enabled and built by default in both the Linux and FreeBSD builds.

11.15.4 Initialization

To use the PMD in an application, user must:

- Call rte_vdev_init("crypto_null") within the application.
- Use –vdev="crypto_null" in the EAL options, which will call rte_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 1 -n 4 --vdev="crypto_null,socket_id=0,max_nb_sessions=128" \
-- -p 1 --cdev SW --chain CIPHER_ONLY --cipher_algo "null"
```

11.16 Cryptodev Scheduler Poll Mode Driver Library

Scheduler PMD is a software crypto PMD, which has the capabilities of attaching hardware and/or software cryptodevs, and distributes ingress crypto ops among them in a certain manner.

Fig. 11.1: Cryptodev Scheduler Overview

The Cryptodev Scheduler PMD library (**librte_pmd_crypto_scheduler**) acts as a software crypto PMD and shares the same API provided by librte_cryptodev. The PMD supports attaching multiple crypto PMDs, software or hardware, as slaves, and distributes the crypto workload to them with certain behavior. The behaviors are categorizes as different "modes". Basically, a scheduling mode defines certain actions for scheduling crypto ops to its slaves.

The librte_pmd_crypto_scheduler library exports a C API which provides an API for attaching/detaching slaves, set/get scheduling modes, and enable/disable crypto ops reordering.

11.16.1 Limitations

- Sessionless crypto operation is not supported
- OOP crypto operation is not supported when the crypto op reordering feature is enabled.

11.16.2 Installation

To build DPDK with CRYTPO_SCHEDULER_PMD the user is required to set CONFIG_RTE_LIBRTE_PMD_CRYPTO_SCHEDULER=y in config/common_base, and recompile DPDK

11.16.3 Initialization

To use the PMD in an application, user must:

- Call rte_vdev_init("crypto_scheduler") within the application.
- Use –vdev="crypto_scheduler" in the EAL options, which will call rte_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_sessions: Specify the maximum number of sessions that can be created. This value may be overwritten internally if there are too many devices are attached.
- slave: If a cryptodev has been initialized with specific name, it can be attached to the scheduler using this parameter, simply filling the name here. Multiple cryptodevs can be attached initially by presenting this parameter multiple times.

- mode: Specify the scheduling mode of the PMD. The supported scheduling mode parameter values are specified in the "Cryptodev Scheduler Modes Overview" section.
- mode_param: Specify the mode-specific parameter. Some scheduling modes may be initialized with specific parameters other than the default ones, such as the **threshold** packet size of **packet-size-distr** mode. This parameter fulfills the purpose.
- ordering: Specify the status of the crypto operations ordering feature. The value of this parameter can be "enable" or "disable". This feature is disabled by default.

Example:

```
... --vdev "crypto_aesni_mb0,name=aesni_mb_1" --vdev "crypto_aesni_mb1,name=aesni_mb_2" --vdev
↪"crypto_scheduler,slave=aesni_mb_1,slave=aesni_mb_2" ...
```

Note:

- The scheduler cryptodev cannot be started unless the scheduling mode is set and at least one slave is attached. Also, to configure the scheduler in the run-time, like attach/detach slave(s), change scheduling mode, or enable/disable crypto op ordering, one should stop the scheduler first, otherwise an error will be returned.
- The crypto op reordering feature requires using the userdata field of every mbuf to be processed to store temporary data. By the end of processing, the field is set to pointing to NULL, any previously stored value of this field will be lost.

11.16.4 Cryptodev Scheduler Modes Overview

Currently the Crypto Scheduler PMD library supports following modes of operation:

- **CDEV_SCHED_MODE_ROUNDROBIN:**

  Initialization mode parameter: **round-robin**

  Round-robin mode, which distributes the enqueued burst of crypto ops among its slaves in a round-robin manner. This mode may help to fill the throughput gap between the physical core and the existing cryptodevs to increase the overall performance.

- **CDEV_SCHED_MODE_PKT_SIZE_DISTR:**

  Initialization mode parameter: **packet-size-distr**

  Packet-size based distribution mode, which works with 2 slaves, the primary slave and the secondary slave, and distributes the enqueued crypto operations to them based on their data lengths. A crypto operation will be distributed to the primary slave if its data length is equal to or bigger than the designated threshold, otherwise it will be handled by the secondary slave.

  A typical usecase in this mode is with the QAT cryptodev as the primary and a software cryptodev as the secondary slave. This may help applications to process additional crypto workload than what the QAT cryptodev can handle on its own, by making use of the available CPU cycles to deal with smaller crypto workloads.

  The threshold is set to 128 bytes by default. It can be updated by calling function **rte_cryptodev_scheduler_option_set**. The parameter of **option_type** must be **CDEV_SCHED_OPTION_THRESHOLD** and **option** should point to a

rte_cryptodev_scheduler_threshold_option structure filled with appropriate threshold value. Please NOTE this threshold has be a power-of-2 unsigned integer. It is possible to use **mode_param** initialization parameter to achieve the same purpose. For example:

… –vdev "crypto_scheduler,mode=packet-size-distr,mode_param=threshold:512" …

The above parameter will overwrite the threshold value to 512.

- **CDEV_SCHED_MODE_FAILOVER:**

  Initialization mode parameter: **fail-over**

  Fail-over mode, which works with 2 slaves, the primary slave and the secondary slave. In this mode, the scheduler will enqueue the incoming crypto operation burst to the primary slave. When one or more crypto operations fail to be enqueued, then they will be enqueued to the secondary slave.

- **CDEV_SCHED_MODE_MULTICORE:**

  Initialization mode parameter: **multi-core**

  Multi-core mode, which distributes the workload with several (up to eight) worker cores. The enqueued bursts are distributed among the worker cores in a round-robin manner. If scheduler cannot enqueue entire burst to the same worker, it will enqueue the remaining operations to the next available worker. For pure small packet size (64 bytes) traffic however the multi-core mode is not an optimal solution, as it doesn't give significant per-core performance improvement. For mixed traffic (IMIX) the optimal number of worker cores is around 2-3. For large packets (1.5 kbytes) scheduler shows linear scaling in performance up to eight cores. Each worker uses its own slave cryptodev. Only software cryptodevs are supported. Only the same type of cryptodevs should be used concurrently.

  The multi-core mode uses one extra parameter:

  - corelist: Semicolon-separated list of logical cores to be used as workers. The number of worker cores should be equal to the number of slave cryptodevs. These cores should be present in EAL core list parameter and should not be used by the application or any other process.

  Example:

  … –vdev "crypto_aesni_mb1,name=aesni_mb_1" –vdev "crypto_aesni_mb_pmd2,name=aesni_mb_2" –vdev "crypto_scheduler,slave=aesni_mb_1,slave=aesni_mb_2,mode=multi-core,corelist=23;24" …

11.17 SNOW 3G Crypto Poll Mode Driver

The SNOW3G PMD (**librte_snow3g_zuc**) provides poll mode crypto driver support for utilizing Intel IPSec Multi-buffer library which implements F8 and F8 functions for SNOW 3G UEA2 cipher and UIA2 hash algorithms.

11.17.1 Features

SNOW 3G PMD has support for:

Cipher algorithm:

- RTE_CRYPTO_CIPHER_SNOW3G_UEA2

Authentication algorithm:

- RTE_CRYPTO_AUTH_SNOW3G_UIA2

11.17.2 Limitations

- Chained mbufs are not supported.
- SNOW 3G (UIA2) supported only if hash offset field is byte-aligned.
- In-place bit-level operations for SNOW 3G (UEA2) are not supported (if length and/or offset of data to be ciphered is not byte-aligned).

11.17.3 Installation

To build DPDK with the SNOW3G_PMD the user is required to download the multi-buffer library from here and compile it on their user system before building DPDK. The latest version of the library supported by this PMD is v0.54, which can be downloaded from https://github.com/01org/intel-ipsec-mb/archive/v0.54.zip.

After downloading the library, the user needs to unpack and compile it on their system before building DPDK:

```
make
make install
```

The library requires NASM to be built. Depending on the library version, it might require a minimum NASM version (e.g. v0.54 requires at least NASM 2.14).

NASM is packaged for different OS. However, on some OS the version is too old, so a manual installation is required. In that case, NASM can be downloaded from NASM website. Once it is downloaded, extract it and follow these steps:

```
./configure
make
make install
```

Note: Compilation of the Multi-Buffer library is broken when GCC < 5.0, if library <= v0.53. If a lower GCC version than 5.0, the workaround proposed by the following link should be used: https://github.com/intel/intel-ipsec-mb/issues/40.

As a reference, the following table shows a mapping between the past DPDK versions and the external crypto libraries supported by them:

Table 11.10: DPDK and external crypto library version compatibility

DPDK version	Crypto library version
16.04 - 19.11	LibSSO SNOW3G
20.02+	Multi-buffer library 0.53 - 0.54

11.17.4 Initialization

In order to enable this virtual crypto PMD, user must:

- Build the multi buffer library (explained in Installation section).
- Build DPDK as follows:

```
make config T=x86_64-native-linux-gcc
sed -i 's,\(CONFIG_RTE_LIBRTE_PMD_SNOW3G\)=n,\1=y,' build/.config
make
```

To use the PMD in an application, user must:

- Call rte_vdev_init("crypto_snow3g") within the application.
- Use –vdev="crypto_snow3g" in the EAL options, which will call rte_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 1 -n 4 --vdev="crypto_snow3g,socket_id=0,max_nb_sessions=128" \
-- -p 1 --cdev SW --chain CIPHER_ONLY --cipher_algo "snow3g-uea2"
```

11.18 Intel(R) QuickAssist (QAT) Crypto Poll Mode Driver

QAT documentation consists of three parts:

- Details of the symmetric and asymmetric crypto services below.
- Details of the *compression service* in the compressdev drivers section.
- Details of building the common QAT infrastructure and the PMDs to support the above services. See *Building PMDs on QAT* below.

11.18.1 Symmetric Crypto Service on QAT

The QAT symmetric crypto PMD (hereafter referred to as *QAT SYM [PMD]*) provides poll mode crypto driver support for the following hardware accelerator devices:

- `Intel QuickAssist Technology DH895xCC`
- `Intel QuickAssist Technology C62x`
- `Intel QuickAssist Technology C3xxx`
- `Intel QuickAssist Technology D15xx`
- `Intel QuickAssist Technology P5xxx`

Features

The QAT SYM PMD has support for:

Cipher algorithms:

- `RTE_CRYPTO_CIPHER_3DES_CBC`
- `RTE_CRYPTO_CIPHER_3DES_CTR`
- `RTE_CRYPTO_CIPHER_AES128_CBC`
- `RTE_CRYPTO_CIPHER_AES192_CBC`
- `RTE_CRYPTO_CIPHER_AES256_CBC`
- `RTE_CRYPTO_CIPHER_AES128_CTR`
- `RTE_CRYPTO_CIPHER_AES192_CTR`
- `RTE_CRYPTO_CIPHER_AES256_CTR`
- `RTE_CRYPTO_CIPHER_AES_XTS`
- `RTE_CRYPTO_CIPHER_SNOW3G_UEA2`
- `RTE_CRYPTO_CIPHER_NULL`
- `RTE_CRYPTO_CIPHER_KASUMI_F8`
- `RTE_CRYPTO_CIPHER_DES_CBC`
- `RTE_CRYPTO_CIPHER_AES_DOCSISBPI`
- `RTE_CRYPTO_CIPHER_DES_DOCSISBPI`
- `RTE_CRYPTO_CIPHER_ZUC_EEA3`

Hash algorithms:

- `RTE_CRYPTO_AUTH_SHA1`
- `RTE_CRYPTO_AUTH_SHA1_HMAC`
- `RTE_CRYPTO_AUTH_SHA224`
- `RTE_CRYPTO_AUTH_SHA224_HMAC`
- `RTE_CRYPTO_AUTH_SHA256`

- RTE_CRYPTO_AUTH_SHA256_HMAC
- RTE_CRYPTO_AUTH_SHA384
- RTE_CRYPTO_AUTH_SHA384_HMAC
- RTE_CRYPTO_AUTH_SHA512
- RTE_CRYPTO_AUTH_SHA512_HMAC
- RTE_CRYPTO_AUTH_AES_XCBC_MAC
- RTE_CRYPTO_AUTH_SNOW3G_UIA2
- RTE_CRYPTO_AUTH_MD5_HMAC
- RTE_CRYPTO_AUTH_NULL
- RTE_CRYPTO_AUTH_KASUMI_F9
- RTE_CRYPTO_AUTH_AES_GMAC
- RTE_CRYPTO_AUTH_ZUC_EIA3
- RTE_CRYPTO_AUTH_AES_CMAC

Supported AEAD algorithms:

- RTE_CRYPTO_AEAD_AES_GCM
- RTE_CRYPTO_AEAD_AES_CCM

Supported Chains

All the usual chains are supported and also some mixed chains:

Table 11.11: Supported hash-cipher chains for wireless digest-encrypted cases

Cipher algorithm	NULL AUTH	SNOW3G UIA2	ZUC EIA3	AES CMAC
NULL CIPHER	Y	2&3	2&3	Y
SNOW3G UEA2	2&3	Y	2&3	2&3
ZUC EEA3	2&3	2&3	2&3	2&3
AES CTR	Y	2&3	2&3	Y

- The combinations marked as "Y" are supported on all QAT hardware versions.
- The combinations marked as "2&3" are supported on GEN2/GEN3 QAT hardware only.

Limitations

- Only supports the session-oriented API implementation (session-less APIs are not supported).
- SNOW 3G (UEA2), KASUMI (F8) and ZUC (EEA3) supported only if cipher length and offset fields are byte-multiple.
- SNOW 3G (UIA2) and ZUC (EIA3) supported only if hash length and offset fields are byte-multiple.
- No BSD support as BSD QAT kernel driver not available.

- ZUC EEA3/EIA3 is not supported by dh895xcc devices
- Maximum additional authenticated data (AAD) for GCM is 240 bytes long and must be passed to the device in a buffer rounded up to the nearest block-size multiple (x16) and padded with zeros.
- Queue-pairs are thread-safe on Intel CPUs but Queues are not (that is, within a single queue-pair all enqueues to the TX queue must be done from one thread and all dequeues from the RX queue must be done from one thread, but enqueues and dequeues may be done in different threads.)
- A GCM limitation exists, but only in the case where there are multiple generations of QAT devices on a single platform. To optimise performance, the GCM crypto session should be initialised for the device generation to which the ops will be enqueued. Specifically if a GCM session is initialised on a GEN2 device, but then attached to an op enqueued to a GEN3 device, it will work but cannot take advantage of hardware optimisations in the GEN3 device. And if a GCM session is initialised on a GEN3 device, then attached to an op sent to a GEN1/GEN2 device, it will not be enqueued to the device and will be marked as failed. The simplest way to mitigate this is to use the bdf whitelist to avoid mixing devices of different generations in the same process if planning to use for GCM.
- The mixed algo feature on GEN2 is not supported by all kernel drivers. Check the notes under the Available Kernel Drivers table below for specific details.

Extra notes on KASUMI F9

When using KASUMI F9 authentication algorithm, the input buffer must be constructed according to the 3GPP KASUMI specification (section 4.4, page 13). The input buffer has to have COUNT (4 bytes), FRESH (4 bytes), MESSAGE and DIRECTION (1 bit) concatenated. After the DIRECTION bit, a single '1' bit is appended, followed by between 0 and 7 '0' bits, so that the total length of the buffer is multiple of 8 bits. Note that the actual message can be any length, specified in bits.

Once this buffer is passed this way, when creating the crypto operation, length of data to authenticate "op.sym.auth.data.length" must be the length of all the items described above, including the padding at the end. Also, offset of data to authenticate "op.sym.auth.data.offset" must be such that points at the start of the COUNT bytes.

11.18.2 Asymmetric Crypto Service on QAT

The QAT asymmetric crypto PMD (hereafter referred to as *QAT ASYM [PMD]*) provides poll mode crypto driver support for the following hardware accelerator devices:

- `Intel QuickAssist Technology DH895xCC`
- `Intel QuickAssist Technology C62x`
- `Intel QuickAssist Technology C3xxx`
- `Intel QuickAssist Technology D15xx`
- `Intel QuickAssist Technology P5xxx`

The QAT ASYM PMD has support for:

- `RTE_CRYPTO_ASYM_XFORM_MODEX`
- `RTE_CRYPTO_ASYM_XFORM_MODINV`

Limitations

- Big integers longer than 4096 bits are not supported.
- Queue-pairs are thread-safe on Intel CPUs but Queues are not (that is, within a single queue-pair all enqueues to the TX queue must be done from one thread and all dequeues from the RX queue must be done from one thread, but enqueues and dequeues may be done in different threads.)
- RSA-2560, RSA-3584 are not supported

11.18.3 Building PMDs on QAT

A QAT device can host multiple acceleration services:

- symmetric cryptography
- data compression
- asymmetric cryptography

These services are provided to DPDK applications via PMDs which register to implement the corresponding cryptodev and compressdev APIs. The PMDs use common QAT driver code which manages the QAT PCI device. They also depend on a QAT kernel driver being installed on the platform, see *Dependency on the QAT kernel driver* below.

Configuring and Building the DPDK QAT PMDs

Further information on configuring, building and installing DPDK is described *here*.

Quick instructions for QAT cryptodev PMD are as follows:

```
cd to the top-level DPDK directory
make defconfig
sed -i 's,\(CONFIG_RTE_LIBRTE_PMD_QAT_SYM\)=n,\1=y,' build/.config
or/and
sed -i 's,\(CONFIG_RTE_LIBRTE_PMD_QAT_ASYM\)=n,\1=y,' build/.config
make
```

Quick instructions for QAT compressdev PMD are as follows:

```
cd to the top-level DPDK directory
make defconfig
make
```

Build Configuration

These are the build configuration options affecting QAT, and their default values:

```
CONFIG_RTE_LIBRTE_PMD_QAT=y
CONFIG_RTE_LIBRTE_PMD_QAT_SYM=n
CONFIG_RTE_LIBRTE_PMD_QAT_ASYM=n
CONFIG_RTE_PMD_QAT_MAX_PCI_DEVICES=48
CONFIG_RTE_PMD_QAT_COMP_IM_BUFFER_SIZE=65536
```

CONFIG_RTE_LIBRTE_PMD_QAT must be enabled for any QAT PMD to be built.

Both QAT SYM PMD and QAT ASYM PMD have an external dependency on libcrypto, so are not built by default. CONFIG_RTE_LIBRTE_PMD_QAT_SYM/ASYM should be enabled to build them.

The QAT compressdev PMD has no external dependencies, so needs no configuration options and is built by default.

The number of VFs per PF varies - see table below. If multiple QAT packages are installed on a platform then CONFIG_RTE_PMD_QAT_MAX_PCI_DEVICES should be adjusted to the number of VFs which the QAT common code will need to handle.

Note: There are separate config items (not QAT-specific) for max cryptodevs CONFIG_RTE_CRYPTO_MAX_DEVS and max compressdevs CONFIG_RTE_COMPRESS_MAX_DEVS, if necessary these should be adjusted to handle the total of QAT and other devices which the process will use. In particular for crypto, where each QAT VF may expose two crypto devices, sym and asym, it may happen that the number of devices will be bigger than MAX_DEVS and the process will show an error during PMD initialisation. To avoid this problem CONFIG_RTE_CRYPTO_MAX_DEVS may be increased or -w, pci-whitelist domain:bus:devid:func option may be used.

QAT compression PMD needs intermediate buffers to support Deflate compression with Dynamic Huffman encoding. CONFIG_RTE_PMD_QAT_COMP_IM_BUFFER_SIZE specifies the size of a single buffer, the PMD will allocate a multiple of these, plus some extra space for associated meta-data. For GEN2 devices, 20 buffers are allocated while for GEN1 devices, 12 buffers are allocated, plus 1472 bytes overhead.

Note: If the compressed output of a Deflate operation using Dynamic Huffman Encoding is too big to fit in an intermediate buffer, then the operation will be split into smaller operations and their results will be merged afterwards. This is not possible if any checksum calculation was requested - in such case the code falls back to fixed compression. To avoid this less performant case, applications should configure the intermediate buffer size to be larger than the expected input data size (compressed output size is usually unknown, so the only option is to make larger than the input size).

Running QAT PMD with minimum threshold for burst size

If only a small number or packets can be enqueued. Each enqueue causes an expensive MMIO write. These MMIO write occurrences can be optimised by setting any of the following parameters:

- qat_sym_enq_threshold
- qat_asym_enq_threshold
- qat_comp_enq_threshold

When any of these parameters is set rte_cryptodev_enqueue_burst function will return 0 (thereby avoiding an MMIO) if the device is congested and number of packets possible to enqueue is smaller. To use this feature the user must set the parameter on process start as a device additional parameter:

```
-w 03:01.1,qat_sym_enq_threshold=32,qat_comp_enq_threshold=16
```

All parameters can be used with the same device regardless of order. Parameters are separated by comma. When the same parameter is used more than once first occurrence of the parameter is used. Maximum threshold that can be set is 32.

Device and driver naming

- The qat cryptodev symmetric crypto driver name is "crypto_qat".
- The qat cryptodev asymmetric crypto driver name is "crypto_qat_asym".

The "rte_cryptodev_devices_get()" returns the devices exposed by either of these drivers.

- Each qat sym crypto device has a unique name, in format "<pci bdf>_<service>", e.g. "0000:41:01.0_qat_sym".
- Each qat asym crypto device has a unique name, in format "<pci bdf>_<service>", e.g. "0000:41:01.0_qat_asym". This name can be passed to "rte_cryptodev_get_dev_id()" to get the device_id.

Note: The cryptodev driver name is passed to the dpdk-test-crypto-perf tool in the "-devtype" parameter.

The qat crypto device name is in the format of the slave parameter passed to the crypto scheduler.

- The qat compressdev driver name is "compress_qat". The rte_compressdev_devices_get() returns the devices exposed by this driver.
- Each qat compression device has a unique name, in format <pci bdf>_<service>, e.g. "0000:41:01.0_qat_comp". This name can be passed to rte_compressdev_get_dev_id() to get the device_id.

Dependency on the QAT kernel driver

To use QAT an SRIOV-enabled QAT kernel driver is required. The VF devices created and initialised by this driver will be used by the QAT PMDs.

Instructions for installation are below, but first an explanation of the relationships between the PF/VF devices and the PMDs visible to DPDK applications.

Each QuickAssist PF device exposes a number of VF devices. Each VF device can enable one symmetric cryptodev PMD and/or one asymmetric cryptodev PMD and/or one compressdev PMD. These QAT PMDs share the same underlying device and pci-mgmt code, but are enumerated independently on their respective APIs and appear as independent devices to applications.

Note: Each VF can only be used by one DPDK process. It is not possible to share the same VF across multiple processes, even if these processes are using different acceleration services.

Conversely one DPDK process can use one or more QAT VFs and can expose both cryptodev and compressdev instances on each of those VFs.

Available kernel drivers

Kernel drivers for each device for each service are listed in the following table. (Scroll right to see the full table)

Table 11.12: QAT device generations, devices and drivers

S	A	C	Gen	Device	Driver/ver	Kernel Module	Pci Driver	PF Did	#PFs	VF Did	VFs/PF
Yes	No	No	1	DH895xCC	Linux/4.4+	qat_dh895xcc	dh895xcc	435	1	443	32
Yes	Yes	No	“	“	01.org/4.2.0+	“	“	“	“	“	“
Yes	Yes	Yes	“	“	01.org/4.3.0+	“	“	“	“	“	“
Yes	No	No	2	C62x	linux/4.5+	qat_c62x	c6xx	37c8	3	37c9	16
Yes	Yes	Yes	“	“	01.org/4.2.0+	“	“	“	“	“	“
Yes	No	No	2	C3xxx	linux/4.5+	qat_c3xxx	c3xxx	19e2	1	19e3	16
Yes	Yes	Yes	“	“	01.org/4.2.0+	“	“	“	“	“	“
Yes	No	No	2	D15xx	p	qat_d15xx	d15xx	6f54	1	6f55	16
Yes	No	No	3	P5xxx	p	qat_p5xxx	p5xxx	18a0	1	18a1	128

- Note: Symmetric mixed crypto algorithms feature on Gen 2 works only with 01.org driver version 4.9.0+

The first 3 columns indicate the service:

- S = Symmetric crypto service (via cryptodev API)
- A = Asymmetric crypto service (via cryptodev API)
- C = Compression service (via compressdev API)

The `Driver` column indicates either the Linux kernel version in which support for this device was introduced or a driver available on Intel's 01.org website. There are both linux in-tree and 01.org kernel drivers available for some devices. p = release pending.

If you are running on a kernel which includes a driver for your device, see *Installation using kernel.org driver* below. Otherwise see *Installation using 01.org QAT driver*.

Installation using kernel.org driver

The examples below are based on the C62x device, if you have a different device use the corresponding values in the above table.

In BIOS ensure that SRIOV is enabled and either:

- Disable VT-d or
- Enable VT-d and set `"intel_iommu=on iommu=pt"` in the grub file.

Check that the QAT driver is loaded on your system, by executing:

```
lsmod | grep qa
```

You should see the kernel module for your device listed, e.g.:

```
qat_c62x               5626  0
intel_qat              82336  1 qat_c62x
```

Next, you need to expose the Virtual Functions (VFs) using the sysfs file system.

First find the BDFs (Bus-Device-Function) of the physical functions (PFs) of your device, e.g.:

```
lspci -d:37c8
```

You should see output similar to:

```
1a:00.0 Co-processor: Intel Corporation Device 37c8
3d:00.0 Co-processor: Intel Corporation Device 37c8
3f:00.0 Co-processor: Intel Corporation Device 37c8
```

Enable the VFs for each PF by echoing the number of VFs per PF to the pci driver:

```
echo 16 > /sys/bus/pci/drivers/c6xx/0000:1a:00.0/sriov_numvfs
echo 16 > /sys/bus/pci/drivers/c6xx/0000:3d:00.0/sriov_numvfs
echo 16 > /sys/bus/pci/drivers/c6xx/0000:3f:00.0/sriov_numvfs
```

Check that the VFs are available for use. For example `lspci -d:37c9` should list 48 VF devices available for a `C62x` device.

To complete the installation follow the instructions in *Binding the available VFs to the DPDK UIO driver*.

Note: If the QAT kernel modules are not loaded and you see an error like `Failed to load MMP firmware qat_895xcc_mmp.bin` in kernel logs, this may be as a result of not using a distribution, but just updating the kernel directly.

Download firmware from the kernel firmware repo.

Copy qat binaries to `/lib/firmware`:

```
cp qat_895xcc.bin /lib/firmware
cp qat_895xcc_mmp.bin /lib/firmware
```

Change to your linux source root directory and start the qat kernel modules:

```
insmod ./drivers/crypto/qat/qat_common/intel_qat.ko
insmod ./drivers/crypto/qat/qat_dh895xcc/qat_dh895xcc.ko
```

Note: If you see the following warning in `/var/log/messages` it can be ignored: `IOMMU should be enabled for SR-IOV to work correctly`.

Installation using 01.org QAT driver

Download the latest QuickAssist Technology Driver from 01.org. Consult the *Getting Started Guide* at the same URL for further information.

The steps below assume you are:

- Building on a platform with one `C62x` device.
- Using package `qat1.7.l.4.2.0-000xx.tar.gz`.
- On Fedora26 kernel `4.11.11-300.fc26.x86_64`.

In the BIOS ensure that SRIOV is enabled and VT-d is disabled.

Uninstall any existing QAT driver, for example by running:

- `./installer.sh uninstall` in the directory where originally installed.

Build and install the SRIOV-enabled QAT driver:

```
mkdir /QAT
cd /QAT

# Copy the package to this location and unpack
tar zxof qat1.7.l.4.2.0-000xx.tar.gz

./configure --enable-icp-sriov=host
make install
```

You can use `cat /sys/kernel/debug/qat<your device type and bdf>/version/fw` to confirm the driver is correctly installed and is using firmware version 4.2.0. You can use `lspci -d:37c9` to confirm the presence of the 16 VF devices available per `C62x` PF.

Confirm the driver is correctly installed and is using firmware version 4.2.0:

```
cat /sys/kernel/debug/qat<your device type and bdf>/version/fw
```

Confirm the presence of 48 VF devices - 16 per PF:

```
lspci -d:37c9
```

To complete the installation - follow instructions in *Binding the available VFs to the DPDK UIO driver*.

Note: If using a later kernel and the build fails with an error relating to `strict_stroul` not being available apply the following patch:

```
/QAT/QAT1.6/quickassist/utilities/downloader/Target_CoreLibs/uclo/include/linux/uclo_platform.h
+ #if LINUX_VERSION_CODE >= KERNEL_VERSION(3,18,5)
+ #define STR_TO_64(str, base, num, endPtr) {endPtr=NULL; if (kstrtoul((str), (base), (num)))␣
↪printk("Error strtoull convert %s\n", str); }
+ #else
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,38)
#define STR_TO_64(str, base, num, endPtr) {endPtr=NULL; if (strict_strtoull((str), (base),␣
↪(num))) printk("Error strtoull convert %s\n", str); }
#else
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,25)
#define STR_TO_64(str, base, num, endPtr) {endPtr=NULL; strict_strtoll((str), (base), (num));}
#else
#define STR_TO_64(str, base, num, endPtr)                                    \
     do {                                                                    \
           if (str[0] == '-')                                                \
           {                                                                 \
                *(num) = -(simple_strtoull((str+1), &(endPtr), (base)));     \
           }else {                                                           \
                *(num) = simple_strtoull((str), &(endPtr), (base));          \
           }                                                                 \
     } while(0)
+ #endif
#endif
#endif
```

Note: If the build fails due to missing header files you may need to do following:

```
sudo yum install zlib-devel
sudo yum install openssl-devel
sudo yum install libudev-devel
```

Note: If the build or install fails due to mismatching kernel sources you may need to do the following:

```
sudo yum install kernel-headers-`uname -r`
sudo yum install kernel-src-`uname -r`
sudo yum install kernel-devel-`uname -r`
```

Binding the available VFs to the DPDK UIO driver

Unbind the VFs from the stock driver so they can be bound to the uio driver.

For an Intel(R) QuickAssist Technology DH895xCC device

The unbind command below assumes `BDFs` of `03:01.00-03:04.07`, if your VFs are different adjust the unbind command below:

```
for device in $(seq 1 4); do \
    for fn in $(seq 0 7); do \
        echo -n 0000:03:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:03\:0${device}.${fn}/driver/unbind; \
    done; \
done
```

For an Intel(R) QuickAssist Technology C62x device

The unbind command below assumes `BDFs` of `1a:01.00-1a:02.07`, `3d:01.00-3d:02.07` and `3f:01.00-3f:02.07`, if your VFs are different adjust the unbind command below:

```
for device in $(seq 1 2); do \
    for fn in $(seq 0 7); do \
        echo -n 0000:1a:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:1a\:0${device}.${fn}/driver/unbind; \

        echo -n 0000:3d:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:3d\:0${device}.${fn}/driver/unbind; \

        echo -n 0000:3f:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:3f\:0${device}.${fn}/driver/unbind; \
    done; \
done
```

For Intel(R) QuickAssist Technology C3xxx or D15xx device

The unbind command below assumes `BDFs` of `01:01.00-01:02.07`, if your VFs are different adjust the unbind command below:

```
for device in $(seq 1 2); do \
    for fn in $(seq 0 7); do \
        echo -n 0000:01:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:01\:0${device}.${fn}/driver/unbind; \
    done; \
done
```

Bind to the DPDK uio driver

Install the DPDK igb_uio driver, bind the VF PCI Device id to it and use lspci to confirm the VF devices are now in use by igb_uio kernel driver, e.g. for the C62x device:

```
cd to the top-level DPDK directory
modprobe uio
insmod ./build/kmod/igb_uio.ko
echo "8086 37c9" > /sys/bus/pci/drivers/igb_uio/new_id
lspci -vvd:37c9
```

Another way to bind the VFs to the DPDK UIO driver is by using the `dpdk-devbind.py` script:

```
cd to the top-level DPDK directory
./usertools/dpdk-devbind.py -b igb_uio 0000:03:01.1
```

Testing

QAT SYM crypto PMD can be tested by running the test application:

```
make defconfig
make -j
cd ./build/app
./test -l1 -n1 -w <your qat bdf>
RTE>>cryptodev_qat_autotest
```

QAT ASYM crypto PMD can be tested by running the test application:

```
make defconfig
make -j
cd ./build/app
./test -l1 -n1 -w <your qat bdf>
RTE>>cryptodev_qat_asym_autotest
```

QAT compression PMD can be tested by running the test application:

```
make defconfig
sed -i 's,\(CONFIG_RTE_COMPRESSDEV_TEST\)=n,\1=y,' build/.config
make -j
cd ./build/app
./test -l1 -n1 -w <your qat bdf>
RTE>>compressdev_autotest
```

Debugging

There are 2 sets of trace available via the dynamic logging feature:

- pmd.qat_dp exposes trace on the data-path.
- pmd.qat_general exposes all other trace.

pmd.qat exposes both sets of traces. They can be enabled using the log-level option (where 8=maximum log level) on the process cmdline, e.g. using any of the following:

```
--log-level="pmd.qat_general,8"
--log-level="pmd.qat_dp,8"
--log-level="pmd.qat,8"
```

Note: The global RTE_LOG_DP_LEVEL overrides data-path trace so must be set to RTE_LOG_DEBUG to see all the trace. This variable is in config/rte_config.h for meson build and config/common_base for gnu make. Also the dynamic global log level overrides both sets of trace, so e.g. no QAT trace would display in this case:

```
--log-level="7" --log-level="pmd.qat_general,8"
```

11.19 Virtio Crypto Poll Mode Driver

The virtio crypto PMD provides poll mode driver support for the virtio crypto device.

11.19.1 Features

The virtio crypto PMD has support for:

Cipher algorithms:

- RTE_CRYPTO_CIPHER_AES_CBC

Hash algorithms:

- RTE_CRYPTO_AUTH_SHA1_HMAC

11.19.2 Limitations

- Only supports the session-oriented API implementation (session-less APIs are not supported).
- Only supports modern mode since virtio crypto conforms to virtio-1.0.
- Only has two types of queues: data queue and control queue. These two queues only support indirect buffers to communication with the virtio backend.
- Only supports AES_CBC cipher only algorithm and AES_CBC with HMAC_SHA1 chaining algorithm since the vhost crypto backend only these algorithms are supported.
- Does not support Link State interrupt.
- Does not support runtime configuration.

11.19.3 Virtio crypto PMD Rx/Tx Callbacks

Rx callbacks:

- `virtio_crypto_pkt_rx_burst`

Tx callbacks:

- `virtio_crypto_pkt_tx_burst`

11.19.4 Installation

Quick instructions are as follows:

Firstly run DPDK vhost crypto sample as a server side and build QEMU with vhost crypto enabled. QEMU can then be started using the following parameters:

```
qemu-system-x86_64 \
[...] \
    -chardev socket,id=charcrypto0,path=/path/to/your/socket \
    -object cryptodev-vhost-user,id=cryptodev0,chardev=charcrypto0 \
    -device virtio-crypto-pci,id=crypto0,cryptodev=cryptodev0
[...]
```

Secondly bind the uio_generic driver for the virtio-crypto device. For example, 0000:00:04.0 is the domain, bus, device and function number of the virtio-crypto device:

```
modprobe uio_pci_generic
echo -n 0000:00:04.0 > /sys/bus/pci/drivers/virtio-pci/unbind
echo "1af4 1054" > /sys/bus/pci/drivers/uio_pci_generic/new_id
```

Finally the front-end virtio crypto PMD driver can be installed:

```
cd to the top-level DPDK directory
sed -i 's,\(CONFIG_RTE_LIBRTE_PMD_VIRTIO_CRYPTO\)=n,\1=y,' config/common_base
make config T=x86_64-native-linux-gcc
make install T=x86_64-native-linux-gcc
```

11.19.5 Tests

The unit test cases can be tested as below:

```
reserve enough huge pages
cd to the top-level DPDK directory
export RTE_TARGET=x86_64-native-linux-gcc
export RTE_SDK=`pwd`
cd to app/test
type the command "make" to compile
run the tests with "./test"
type the command "cryptodev_virtio_autotest" to test
```

The performance can be tested as below:

```
reserve enough huge pages
cd to the top-level DPDK directory
export RTE_TARGET=x86_64-native-linux-gcc
```

(continues on next page)

(continued from previous page)

```
export RTE_SDK=`pwd`
cd to app/test-crypto-perf
type the command "make" to compile
run the tests with the following command:

./dpdk-test-crypto-perf -l 0,1 -- --devtype crypto_virtio \
    --ptest throughput --optype cipher-then-auth --cipher-algo aes-cbc \
    --cipher-op encrypt --cipher-key-sz 16 --auth-algo sha1-hmac \
    --auth-op generate --auth-key-sz 64 --digest-sz 12 \
    --total-ops 100000000 --burst-sz 64 --buffer-sz 2048
```

11.20 ZUC Crypto Poll Mode Driver

The ZUC PMD (**librte_pmd_zuc**) provides poll mode crypto driver support for utilizing Intel IPSec Multi-buffer library which implements F8 and F9 functions for ZUC EEA3 cipher and EIA3 hash algorithms.

11.20.1 Features

ZUC PMD has support for:

Cipher algorithm:

- RTE_CRYPTO_CIPHER_ZUC_EEA3

Authentication algorithm:

- RTE_CRYPTO_AUTH_ZUC_EIA3

11.20.2 Limitations

- Chained mbufs are not supported.
- ZUC (EIA3) supported only if hash offset field is byte-aligned.
- ZUC (EEA3) supported only if cipher length, cipher offset fields are byte-aligned.

11.20.3 Installation

To build DPDK with the ZUC_PMD the user is required to download the multi-buffer library from here and compile it on their user system before building DPDK. The latest version of the library supported by this PMD is v0.54, which can be downloaded from https://github.com/01org/intel-ipsec-mb/archive/v0.54.zip.

After downloading the library, the user needs to unpack and compile it on their system before building DPDK:

```
make
make install
```

The library requires NASM to be built. Depending on the library version, it might require a minimum NASM version (e.g. v0.54 requires at least NASM 2.14).

NASM is packaged for different OS. However, on some OS the version is too old, so a manual installation is required. In that case, NASM can be downloaded from NASM website. Once it is downloaded, extract it and follow these steps:

```
./configure
make
make install
```

Note: Compilation of the Multi-Buffer library is broken when GCC < 5.0, if library <= v0.53. If a lower GCC version than 5.0, the workaround proposed by the following link should be used: https://github.com/intel/intel-ipsec-mb/issues/40.

As a reference, the following table shows a mapping between the past DPDK versions and the external crypto libraries supported by them:

Table 11.13: DPDK and external crypto library version compatibility

DPDK version	Crypto library version
16.11 - 19.11	LibSSO ZUC
20.02+	Multi-buffer library 0.53 - 0.54

11.20.4 Initialization

In order to enable this virtual crypto PMD, user must:

- Build the multi buffer library (explained in Installation section).
- Build DPDK as follows:

```
make config T=x86_64-native-linux-gcc
sed -i 's,\(CONFIG_RTE_LIBRTE_PMD_ZUC\)=n,\1=y,' build/.config
make
```

To use the PMD in an application, user must:

- Call rte_vdev_init("crypto_zuc") within the application.
- Use –vdev="crypto_zuc" in the EAL options, which will call rte_vdev_init() internally.

The following parameters (all optional) can be provided in the previous two calls:

- socket_id: Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).
- max_nb_queue_pairs: Specify the maximum number of queue pairs in the device (8 by default).
- max_nb_sessions: Specify the maximum number of sessions that can be created (2048 by default).

Example:

```
./l2fwd-crypto -l 1 -n 4 --vdev="crypto_zuc,socket_id=0,max_nb_sessions=128" \
-- -p 1 --cdev SW --chain CIPHER_ONLY --cipher_algo "zuc-eea3"
```

CHAPTER

TWELVE

COMPRESSION DEVICE DRIVERS

12.1 Compression Device Supported Functionality Matrices

12.1.1 Supported Feature Flags

Table 12.1: Features availability in compression drivers

Feature	isal	octeontx	qat	zlib
HW Accelerated		Y	Y	
CPU SSE	Y			
CPU AVX	Y			
CPU AVX2	Y			
CPU AVX512	Y			
CPU NEON				
Stateful Compression				
Stateful Decompression			Y	
Pass-through				Y
OOP SGL In SGL Out	Y		Y	
OOP SGL In LB Out	Y		Y	
OOP LB In SGL Out	Y		Y	
Deflate	Y	Y	Y	Y
LZS				
Adler32	Y		Y	
Crc32	Y		Y	
Adler32&Crc32			Y	
Fixed	Y	Y	Y	Y
Dynamic	Y	Y	Y	Y

Note:

- "Pass-through" feature flag refers to the ability of the PMD to let input buffers pass-through it, copying the input to the output, without making any modifications to it (no compression done).
- "OOP SGL In SGL Out" feature flag stands for "Out-of-place Scatter-gather list Input, Scatter-gather list Output", which means PMD supports different scatter-gather styled input and output buffers (i.e. both can consists of multiple segments).
- "OOP SGL In LB Out" feature flag stands for "Out-of-place Scatter-gather list Input, Linear Buffers Output", which means PMD supports input from scatter-gathered styled buffers, outputting linear

buffers (i.e. single segment).

- "OOP LB In SGL Out" feature flag stands for "Out-of-place Linear Buffers Input, Scatter-gather list Output", which means PMD supports input from linear buffer, outputting scatter-gathered styled buffers.

12.2 ISA-L Compression Poll Mode Driver

The ISA-L PMD (**librte_pmd_isal_comp**) provides poll mode compression & decompression driver support for utilizing Intel ISA-L library, which implements the deflate algorithm for both Deflate(compression) and Inflate(decompression).

12.2.1 Features

ISA-L PMD has support for:

Compression/Decompression algorithm:

- DEFLATE

Huffman code type:

- FIXED
- DYNAMIC

Window size support:

- 32K

Checksum:

- CRC32
- ADLER32

To enable a checksum in the driver, the compression and/or decompression xform structure, rte_comp_xform, must be filled with either of the CompressDev checksum flags supported.

```
compress_xform->compress.chksum = RTE_COMP_CHECKSUM_CRC32

decompress_xform->decompress.chksum = RTE_COMP_CHECKSUM_CRC32
```

```
compress_xform->compress.chksum = RTE_COMP_CHECKSUM_ADLER32

decompress_xform->decompress.chksum = RTE_COMP_CHECKSUM_ADLER32
```

If you request a checksum for compression or decompression, the checksum field in the operation structure, `op->output_chksum`, will be filled with the checksum.

Note: For the compression case above, your output buffer will need to be large enough to hold the compressed data plus a scratchpad for the checksum at the end, the scratchpad is 8 bytes for CRC32 and 4 bytes for Adler32.

Level guide:

The ISA-L levels have been mapped to somewhat correspond to the same ZLIB level, i.e. ZLIB L1 gives a compression ratio similar to ISA-L L1. Compressdev level 0 enables "No Compression", which passes the uncompressed data to the output buffer, plus deflate headers. The ISA-L library does not support this, therefore compressdev level 0 is not supported.

The compressdev API has 10 levels, 0-9. ISA-L has 4 levels of compression, 0-3. As a result the level mappings from the API to the PMD are shown below.

Table 12.2: Level mapping from Compressdev to ISA-L PMD.

Compressdev API Level	PMD Functionality	Internal ISA-L Level
0	No compression, Not Supported	—
1	Dynamic (Fast compression)	1
2	Dynamic (Higher compression ratio)	2
3	Dynamic (Best compression ratio)	3 (Level 2 if no AVX512/AVX2)
4	Dynamic (Best compression ratio)	Same as above
5	Dynamic (Best compression ratio)	Same as above
6	Dynamic (Best compression ratio)	Same as above
7	Dynamic (Best compression ratio)	Same as above
8	Dynamic (Best compression ratio)	Same as above
9	Dynamic (Best compression ratio)	Same as above

Note: The above table only shows mapping when API calls for dynamic compression. For fixed compression, regardless of API level, internally ISA-L level 0 is always used.

12.2.2 Limitations

- Compressdev level 0, no compression, is not supported.

12.2.3 Installation

- To build DPDK with Intel's ISA-L library, the user is required to download the library from https://github.com/01org/isa-l.
- Once downloaded, the user needs to build the library, the ISA-L autotools are usually sufficient:

```
./autogen.sh
./configure
```

- make can be used to install the library on their system, before building DPDK:

```
make
sudo make install
```

- To build with meson, the **libisal.pc** file, must be copied into "pkgconfig", e.g. /usr/lib/pkgconfig or /usr/lib64/pkgconfig depending on your system, for meson to find the ISA-L library. The **libisal.pc** is located in library sources:

```
cp isal/libisal.pc /usr/lib/pkgconfig/
```

12.2.4 Initialization

In order to enable this virtual compression PMD, user must:

- Set `CONFIG_RTE_LIBRTE_PMD_ISAL=y` in config/common_base.

To use the PMD in an application, user must:

- Call `rte_vdev_init("compress_isal")` within the application.
- Use `--vdev="compress_isal"` in the EAL options, which will call `rte_vdev_init()` internally.

The following parameter (optional) can be provided in the previous two calls:

- `socket_id:` Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).

12.3 OCTEON TX ZIP Compression Poll Mode Driver

The OCTEON TX ZIP PMD (**librte_pmd_octeontx_zip**) provides poll mode compression & decompression driver for ZIP HW offload device, found in **Cavium OCTEON TX** SoC family.

More information can be found at Cavium, Inc Official Website.

12.3.1 Features

OCTEON TX ZIP PMD has support for:

Compression/Decompression algorithm:

- DEFLATE

Huffman code type:

- FIXED
- DYNAMIC

Window size support:

- 2 to 2^14

12.3.2 Limitations

- Chained mbufs are not supported.

12.3.3 Supported OCTEON TX SoCs

- CN83xx

12.3.4 Steps To Setup Platform

OCTEON TX SDK includes kernel image which provides OCTEON TX ZIP PF driver to manage configuration of ZIPVF device Required version of SDK is "OCTEONTX-SDK-6.2.0-build35" or above.

SDK can be install by using below command. #rpm -ivh OCTEONTX-SDK-6.2.0-build35.x86_64.rpm –force –nodeps It will install OCTEONTX-SDK at following default location /usr/local/Cavium_Networks/OCTEONTX-SDK/

For more information on building and booting linux kernel on OCTEON TX please refer /usr/local/Cavium_Networks/OCTEONTX-SDK/docs/OcteonTX-SDK-UG_6.2.0.pdf.

SDK and related information can be obtained from: Cavium support site.

12.3.5 Installation

Driver Compilation

To compile the OCTEON TX ZIP PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-thunderx-linux-gcc install
```

12.3.6 Initialization

The OCTEON TX zip is exposed as pci device which consists of a set of PCIe VF devices. On EAL initialization, ZIP PCIe VF devices will be probed. To use the PMD in an application, user must:

- run dev_bind script to bind eight ZIP PCIe VFs to the `vfio-pci` driver:

```
./usertools/dpdk-devbind.py -b vfio-pci 0001:04:00.1
./usertools/dpdk-devbind.py -b vfio-pci 0001:04:00.2
./usertools/dpdk-devbind.py -b vfio-pci 0001:04:00.3
./usertools/dpdk-devbind.py -b vfio-pci 0001:04:00.4
./usertools/dpdk-devbind.py -b vfio-pci 0001:04:00.5
./usertools/dpdk-devbind.py -b vfio-pci 0001:04:00.6
./usertools/dpdk-devbind.py -b vfio-pci 0001:04:00.7
./usertools/dpdk-devbind.py -b vfio-pci 0001:04:01.0
```

- The unit test cases can be tested as below:

```
reserve enough huge pages
cd to the top-level DPDK directory
export RTE_TARGET=arm64-thunderx-linux-gcc
export RTE_SDK=`pwd`
cd to app/test
type the command "make" to compile
run the tests with "./test"
type the command "compressdev_autotest" to test
```

12.4 Intel(R) QuickAssist (QAT) Compression Poll Mode Driver

The QAT compression PMD provides poll mode compression & decompression driver support for the following hardware accelerator devices:

- `Intel QuickAssist Technology C62x`
- `Intel QuickAssist Technology C3xxx`
- `Intel QuickAssist Technology DH895x`

12.4.1 Features

QAT compression PMD has support for:

Compression/Decompression algorithm:

- DEFLATE - using Fixed and Dynamic Huffman encoding

Window size support:

- 32K

Checksum generation:

- CRC32, Adler and combined checksum

Stateful operation:

- Decompression only

12.4.2 Limitations

- Compressdev level 0, no compression, is not supported.
- Queue-pairs are thread-safe on Intel CPUs but Queues are not (that is, within a single queue-pair all enqueues to the TX queue must be done from one thread and all dequeues from the RX queue must be done from one thread, but enqueues and dequeues may be done in different threads.)
- No BSD support as BSD QAT kernel driver not available.
- Stateful compression is not supported.

12.4.3 Installation

The QAT compression PMD is built by default with a standard DPDK build.

It depends on a QAT kernel driver, see *Building PMDs on QAT*.

12.5 ZLIB Compression Poll Mode Driver

The ZLIB PMD (**librte_pmd_zlib**) provides poll mode compression & decompression driver based on SW zlib library,

12.5.1 Features

ZLIB PMD has support for:

Compression/Decompression algorithm:

- DEFLATE

Huffman code type:

- FIXED
- DYNAMIC

Window size support:

- Min - 256 bytes
- Max - 32K

12.5.2 Limitations

- Scatter-Gather and Stateful not supported.

12.5.3 Installation

- To build DPDK with ZLIB library, the user is required to download the `libz` library.
- Use following command for installation.
- **For Fedora users::**
  sudo yum install zlib-devel
- **For Ubuntu users::**
  sudo apt-get install zlib1g-dev
- Once downloaded, the user needs to build the library.
- To build from sources download zlib sources from http://zlib.net/ and do following before building DPDK:

```
make
sudo make install
```

12.5.4 Initialization

In order to enable this virtual compression PMD, user must:

- Set `CONFIG_RTE_LIBRTE_PMD_ZLIB=y` in config/common_base.

To use the PMD in an application, user must:

- Call `rte_vdev_init("compress_zlib")` within the application.
- Use `--vdev="compress_zlib"` in the EAL options, which will call `rte_vdev_init()` internally.

The following parameter (optional) can be provided in the previous two calls:

- `socket_id:` Specify the socket where the memory for the device is going to be allocated (by default, socket_id will be the socket where the core that is creating the PMD is running on).

CHAPTER

THIRTEEN

VDPA DEVICE DRIVERS

The following are a list of vDPA (vHost Data Path Acceleration) device drivers, which can be used from an application through vhost API.

13.1 Overview of vDPA Drivers Features

This section explains the supported features that are listed in the table below.

csum
: Device can handle packets with partial checksum.

guest csum
: Guest can handle packets with partial checksum.

mac
: Device has given MAC address.

gso
: Device can handle packets with any GSO type.

guest tso4
: Guest can receive TSOv4.

guest tso6
: Guest can receive TSOv6.

ecn
: Device can receive TSO with ECN.

ufo
: Device can receive UFO.

host tso4
: Device can receive TSOv4.

host tso6
: Device can receive TSOv6.

mrg rxbuf
: Guest can merge receive buffers.

ctrl vq
: Control channel is available.

ctrl rx
Control channel RX mode support.

any layout
Device can handle any descriptor layout.

guest announce
Guest can send gratuitous packets.

mq
Device supports Receive Flow Steering.

version 1
v1.0 compliant.

log all
Device can log all write descriptors (live migration).

indirect desc
Indirect buffer descriptors support.

event idx
Support for avail_idx and used_idx fields.

mtu
Host can advise the guest with its maximum supported MTU.

in_order
Device can use descriptors in ring order.

IOMMU platform
Device support IOMMU addresses.

packed
Device support packed virtio queues.

proto mq
Support the number of queues query.

proto log shmfd
Guest support setting log base.

proto rarp
Host can broadcast a fake RARP after live migration.

proto reply ack
Host support requested operation status ack.

proto host notifier
Host can register memory region based host notifiers.

proto pagefault
Slave expose page-fault FD for migration process.

BSD nic_uio
BSD `nic_uio` module supported.

Linux VFIO
Works with `vfio-pci` kernel module.

Other kdrv
Kernel module other than above ones supported.

ARMv7
Support armv7 architecture.

ARMv8
Support armv8a (64bit) architecture.

Power8
Support PowerPC architecture.

x86-32
Support 32bits x86 architecture.

x86-64
Support 64bits x86 architecture.

Usage doc
Documentation describes usage, In `doc/guides/vdpadevs/`.

Design doc
Documentation describes design. In `doc/guides/vdpadevs/`.

Perf doc
Documentation describes performance values, In `doc/perf/`.

Note: Most of the features capabilities should be provided by the drivers via the next vDPA operations: `get_features` and `get_protocol_features`.

13.2 References

- OASIS: Virtual I/O Device (VIRTIO) Version 1.1
- QEMU: Vhost-user Protocol

13.3 Features Table

Table 13.1: Features availability in vDPA drivers

Feature	i f c v f	m l x 5
csum		Y
guest csum		Y
mac		
gso		
guest tso4		
guest tso6		
ecn		
ufo		
host tso4		Y

continues on next page

Table 13.1 – continued from previous page

Feature	i f c v f	m l x 5
host tso6		Y
mrg rxbuf		
ctrl vq		
ctrl rx		
any layout		Y
guest announce		Y
mq		Y
version 1		Y
log all		Y
indirect desc		
event idx		
mtu		
in_order		
IOMMU platform		
packed		Y
proto mq		Y
proto log shmfd		Y
proto rarp		
proto reply ack		
proto host notifier		Y
proto pagefault		
BSD nic_uio		
Linux VFIO		
Other kdrv		Y
ARMv7		
ARMv8		Y
Power8		Y
x86-32	Y	Y
x86-64	Y	Y
Usage doc		Y
Design doc		Y
Perf doc		

Note: Features marked with "P" are partially supported. Refer to the appropriate driver guide in the following sections for details.

13.4 IFCVF vDPA driver

The IFCVF vDPA (vhost data path acceleration) driver provides support for the Intel FPGA 100G VF (IFCVF). IFCVF's datapath is virtio ring compatible, it works as a HW vhost backend which can send/receive packets to/from virtio directly by DMA. Besides, it supports dirty page logging and device state report/restore, this driver enables its vDPA functionality.

13.4.1 Pre-Installation Configuration

Config File Options

The following option can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_IFC_PMD` (default `y` for linux)

  Toggle compilation of the `librte_pmd_ifc` driver.

13.4.2 IFCVF vDPA Implementation

IFCVF's vendor ID and device ID are same as that of virtio net pci device, with its specific subsystem vendor ID and device ID. To let the device be probed by IFCVF driver, adding "vdpa=1" parameter helps to specify that this device is to be used in vDPA mode, rather than polling mode, virtio pmd will skip when it detects this message. If no this parameter specified, device will not be used as a vDPA device, and it will be driven by virtio pmd.

Different VF devices serve different virtio frontends which are in different VMs, so each VF needs to have its own DMA address translation service. During the driver probe a new container is created for this device, with this container vDPA driver can program DMA remapping table with the VM's memory region information.

The device argument "sw-live-migration=1" will configure the driver into SW assisted live migration mode. In this mode, the driver will set up a SW relay thread when LM happens, this thread will help device to log dirty pages. Thus this mode does not require HW to implement a dirty page logging function block, but will consume some percentage of CPU resource depending on the network throughput. If no this parameter specified, driver will rely on device's logging capability.

Key IFCVF vDPA driver ops

- ifcvf_dev_config: Enable VF data path with virtio information provided by vhost lib, including IOMMU programming to enable VF DMA to VM's memory, VFIO interrupt setup to route HW interrupt to virtio driver, create notify relay thread to translate virtio driver's kick to a MMIO write onto HW, HW queues configuration.

  This function gets called to set up HW data path backend when virtio driver in VM gets ready.

- ifcvf_dev_close: Revoke all the setup in ifcvf_dev_config.

  This function gets called when virtio driver stops device in VM.

To create a vhost port with IFC VF

- Create a vhost socket and assign a VF's device ID to this socket via vhost API. When QEMU vhost connection gets ready, the assigned VF will get configured automatically.

13.4.3 Features

Features of the IFCVF driver are:

- Compatibility with virtio 0.95 and 1.0.
- SW assisted vDPA live migration.

13.4.4 Prerequisites

- Platform with IOMMU feature. IFC VF needs address translation service to Rx/Tx directly with virtio driver in VM.

13.4.5 Limitations

Dependency on vfio-pci

vDPA driver needs to setup VF MSIX interrupts, each queue's interrupt vector is mapped to a callfd associated with a virtio ring. Currently only vfio-pci allows multiple interrupts, so the IFCVF driver is dependent on vfio-pci.

Live Migration with VIRTIO_NET_F_GUEST_ANNOUNCE

IFC VF doesn't support RARP packet generation, virtio frontend supporting VIRTIO_NET_F_GUEST_ANNOUNCE feature can help to do that.

13.5 MLX5 vDPA driver

The MLX5 vDPA (vhost data path acceleration) driver library (**librte_pmd_mlx5_vdpa**) provides support for **Mellanox ConnectX-6**, **Mellanox ConnectX-6 Dx** and **Mellanox BlueField** families of 10/25/40/50/100/200 Gb/s adapters as well as their virtual functions (VF) in SR-IOV context.

Note: Due to external dependencies, this driver is disabled in default configuration of the "make" build. It can be enabled with `CONFIG_RTE_LIBRTE_MLX5_VDPA_PMD=y` or by using "meson" build system which will detect dependencies.

13.5.1 Design

For security reasons and robustness, this driver only deals with virtual memory addresses. The way resources allocations are handled by the kernel, combined with hardware specifications that allow to handle virtual memory addresses directly, ensure that DPDK applications cannot access random physical memory (or memory that does not belong to the current process).

The PMD can use libibverbs and libmlx5 to access the device firmware or directly the hardware components. There are different levels of objects and bypassing abilities to get the best performances:

- Verbs is a complete high-level generic API

- Direct Verbs is a device-specific API
- DevX allows to access firmware objects
- Direct Rules manages flow steering at low-level hardware layer

Enabling librte_pmd_mlx5_vdpa causes DPDK applications to be linked against libibverbs.

A Mellanox mlx5 PCI device can be probed by either net/mlx5 driver or vdpa/mlx5 driver but not in parallel. Hence, the user should decide the driver by the `class` parameter in the device argument list. By default, the mlx5 device will be probed by the net/mlx5 driver.

13.5.2 Supported NICs

- Mellanox® ConnectX®-6 200G MCX654106A-HCAT (2x200G)
- Mellanox® ConnectX®-6 Dx EN 100G MCX623106AN-CDAT (2x100G)
- Mellanox® ConnectX®-6 Dx EN 200G MCX623105AN-VDAT (1x200G)
- Mellanox® BlueField SmartNIC 25G MBF1M332A-ASCAT (2x25G)

13.5.3 Prerequisites

- Mellanox OFED version: **5.0** see *MLX5 poll mode driver* guide for more Mellanox OFED details.

Compilation options

These options can be modified in the `.config` file.

- `CONFIG_RTE_LIBRTE_MLX5_VDPA_PMD` (default **n**)

  Toggle compilation of librte_pmd_mlx5 itself.

- `CONFIG_RTE_IBVERBS_LINK_DLOPEN` (default **n**)

  Build PMD with additional code to make it loadable without hard dependencies on **libibverbs** nor **libmlx5**, which may not be installed on the target system.

  In this mode, their presence is still required for it to run properly, however their absence won't prevent a DPDK application from starting (with `CONFIG_RTE_BUILD_SHARED_LIB` disabled) and they won't show up as missing with `ldd(1)`.

  It works by moving these dependencies to a purpose-built rdma-core "glue" plug-in which must either be installed in a directory whose name is based on `CONFIG_RTE_EAL_PMD_PATH` suffixed with `-glue` if set, or in a standard location for the dynamic linker (e.g. `/lib`) if left to the default empty string ("").

  This option has no performance impact.

- `CONFIG_RTE_IBVERBS_LINK_STATIC` (default **n**)

  Embed static flavor of the dependencies **libibverbs** and **libmlx5** in the PMD shared library or the executable static binary.

Note: For BlueField, target should be set to `arm64-bluefield-linux-gcc`. This will enable `CONFIG_RTE_LIBRTE_MLX5_VDPA_PMD` and set `RTE_CACHE_LINE_SIZE` to 64. Default armv8a configuration of make build and meson build set it to 128 then brings performance degradation.

Run-time configuration

- **ethtool** operations on related kernel interfaces also affect the PMD.
- `class` parameter [string]

  Select the class of the driver that should probe the device. *vdpa* for the mlx5 vDPA driver.

CHAPTER

FOURTEEN

EVENT DEVICE DRIVERS

The following are a list of event device PMDs, which can be used from an application through the eventdev API.

14.1 NXP DPAA Eventdev Driver

The dpaa eventdev is an implementation of the eventdev API, that provides a wide range of the eventdev features. The eventdev relies on a dpaa based platform to perform event scheduling.

More information can be found at NXP Official Website.

14.1.1 Features

The DPAA EVENTDEV implements many features in the eventdev API;

- Hardware based event scheduler
- 4 event ports
- 4 event queues
- Parallel flows
- Atomic flows

14.1.2 Supported DPAA SoCs

- LS1046A/LS1026A
- LS1043A/LS1023A

14.1.3 Prerequisites

See *NXP QorIQ DPAA Board Support Package* for setup information

Currently supported by DPDK:

- NXP SDK **2.0+** or LSDK **18.09+**
- Supported architectures: **arm64 LE**.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

14.1.4 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_PMD_DPAA_EVENTDEV` (default `y`)

  Toggle compilation of the `librte_pmd_dpaa_event` driver.

Driver Compilation

To compile the DPAA EVENTDEV PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-dpaa-linux-gcc install
```

14.1.5 Initialization

The dpaa eventdev is exposed as a vdev device which consists of a set of channels and queues. On EAL initialization, dpaa components will be probed and then vdev device can be created from the application code by

- Invoking `rte_vdev_init("event_dpaa1")` from the application
- Using `--vdev="event_dpaa1"` in the EAL options, which will call rte_vdev_init() internally

Example:

```
./your_eventdev_application --vdev="event_dpaa1"
```

- Use dev arg option `disable_intr=1` to disable the interrupt mode

14.1.6 Limitations

1. DPAA eventdev can not work with DPAA PUSH mode queues configured for ethdev. Please configure export DPAA_NUM_PUSH_QUEUES=0

Platform Requirement

DPAA drivers for DPDK can only work on NXP SoCs as listed in the `Supported DPAA SoCs`.

Port-core Binding

DPAA EVENTDEV driver requires event port 'x' to be used on core 'x'.

14.2 NXP DPAA2 Eventdev Driver

The dpaa2 eventdev is an implementation of the eventdev API, that provides a wide range of the eventdev features. The eventdev relies on a dpaa2 hw to perform event scheduling.

More information can be found at NXP Official Website.

14.2.1 Features

The DPAA2 EVENTDEV implements many features in the eventdev API;

- Hardware based event scheduler
- 8 event ports
- 8 event queues
- Parallel flows
- Atomic flows

14.2.2 Supported DPAA2 SoCs

- LX2160A
- LS2084A/LS2044A
- LS2088A/LS2048A
- LS1088A/LS1048A

14.2.3 Prerequisites

See *NXP QorIQ DPAA2 Board Support Package* for setup information

Currently supported by DPDK:

- NXP SDK **19.09+**.
- MC Firmware version **10.18.0** and higher.
- Supported architectures: **arm64 LE**.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

Note: Some part of fslmc bus code (mc flib - object library) routines are dual licensed (BSD & GPLv2).

14.2.4 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_PMD_DPAA2_EVENTDEV` (default `y`)

  Toggle compilation of the `lrte_pmd_dpaa2_event` driver.

Driver Compilation

To compile the DPAA2 EVENTDEV PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-dpaa-linux-gcc install
```

14.2.5 Initialization

The dpaa2 eventdev is exposed as a vdev device which consists of a set of dpcon devices and dpci devices. On EAL initialization, dpcon and dpci devices will be probed and then vdev device can be created from the application code by

- Invoking `rte_vdev_init("event_dpaa2")` from the application
- Using `--vdev="event_dpaa2"` in the EAL options, which will call rte_vdev_init() internally

Example:

```
./your_eventdev_application --vdev="event_dpaa2"
```

14.2.6 Enabling logs

For enabling logs, use the following EAL parameter:

```
./your_eventdev_application <EAL args> --log-level=pmd.event.dpaa2,<level>
```

Using `eventdev.dpaa2` as log matching criteria, all Event PMD logs can be enabled which are lower than logging `level`.

14.2.7 Limitations

Platform Requirement

DPAA2 drivers for DPDK can only work on NXP SoCs as listed in the `Supported DPAA2 SoCs`.

Port-core binding

DPAA2 EVENTDEV can support only one eventport per core.

14.3 Distributed Software Eventdev Poll Mode Driver

The distributed software event device is an eventdev driver which distributes the task of scheduling events among all the eventdev ports and the lcore threads using them.

14.3.1 Features

Queues

- Atomic
- Parallel
- Single-Link

Ports

- Load balanced (for Atomic, Ordered, Parallel queues)
- Single Link (for single-link queues)

14.3.2 Configuration and Options

The distributed software eventdev is a vdev device, and as such can be created from the application code, or from the EAL command line:

- Call `rte_vdev_init("event_dsw0")` from the application
- Use `--vdev="event_dsw0"` in the EAL options, which will call rte_vdev_init() internally

Example:

```
./your_eventdev_application --vdev="event_dsw0"
```

14.3.3 Limitations

Unattended Ports

The distributed software eventdev uses an internal signaling schema between the ports to achieve load balancing. In order for this to work, the application must perform enqueue and/or dequeue operations on all ports.

Producer-only ports which currently have no events to enqueue should periodically call rte_event_enqueue_burst() with a zero-sized burst.

Ports left unattended for longer periods of time will prevent load balancing, and also cause traffic interruptions on the flows which are in the process of being migrated.

Output Buffering

For efficiency reasons, the distributed software eventdev might not send enqueued events immediately to the destination port, but instead store them in an internal buffer in the source port.

In case no more events are enqueued on a port with buffered events, these events will be sent after the application has performed a number of enqueue and/or dequeue operations.

For explicit flushing, an application may call rte_event_enqueue_burst() with a zero-sized burst.

Priorities

The distributed software eventdev does not support event priorities.

Ordered Queues

The distributed software eventdev does not support the ordered queue type.

"All Types" Queues

The distributed software eventdev does not support queues of type RTE_EVENT_QUEUE_CFG_ALL_TYPES, which allow both atomic, ordered, and parallel events on the same queue.

Dynamic Link/Unlink

The distributed software eventdev does not support calls to rte_event_port_link() or rte_event_port_unlink() after rte_event_dev_start() has been called.

14.4 Software Eventdev Poll Mode Driver

The software eventdev is an implementation of the eventdev API, that provides a wide range of the eventdev features. The eventdev relies on a CPU core to perform event scheduling. This PMD can use the service core library to run the scheduling function, allowing an application to utilize the power of service cores to multiplex other work on the same core if required.

14.4.1 Features

The software eventdev implements many features in the eventdev API;

Queues

- Atomic
- Ordered
- Parallel
- Single-Link

Ports

- Load balanced (for Atomic, Ordered, Parallel queues)
- Single Link (for single-link queues)

Event Priorities

- Each event has a priority, which can be used to provide basic QoS

14.4.2 Configuration and Options

The software eventdev is a vdev device, and as such can be created from the application code, or from the EAL command line:

- Call `rte_vdev_init("event_sw0")` from the application
- Use `--vdev="event_sw0"` in the EAL options, which will call rte_vdev_init() internally

Example:

```
./your_eventdev_application --vdev="event_sw0"
```

Scheduling Quanta

The scheduling quanta sets the number of events that the device attempts to schedule in a single schedule call performed by the service core. Note that is a *hint* only, and that fewer or more events may be scheduled in a given iteration.

The scheduling quanta can be set using a string argument to the vdev create call:

```
--vdev="event_sw0,sched_quanta=64"
```

Credit Quanta

The credit quanta is the number of credits that a port will fetch at a time from the instance's credit pool. Higher numbers will cause less overhead in the atomic credit fetch code, however it also reduces the overall number of credits in the system faster. A balanced number (e.g. 32) ensures that only small numbers of credits are pre-allocated at a time, while also mitigating performance impact of the atomics.

Experimentation with higher values may provide minor performance improvements, at the cost of the whole system having less credits. On the other hand, reducing the quanta may cause measurable performance impact but provide the system with a higher number of credits at all times.

A value of 32 seems a good balance however your specific application may benefit from a higher or reduced quanta size, experimentation is required to verify possible gains.

```
--vdev="event_sw0,credit_quanta=64"
```

14.4.3 Limitations

The software eventdev implementation has a few limitations. The reason for these limitations is usually that the performance impact of supporting the feature would be significant.

"All Types" Queues

The software eventdev does not support creating queues that handle all types of traffic. An eventdev with this capability allows enqueuing Atomic, Ordered and Parallel traffic to the same queue, but scheduling each of them appropriately.

The reason to not allow Atomic, Ordered and Parallel event types in the same queue is that it causes excessive branching in the code to enqueue packets to the queue, causing a significant performance impact.

The `RTE_EVENT_DEV_CAP_QUEUE_ALL_TYPES` flag is not set in the `event_dev_cap` field of the `rte_event_dev_info` struct for the software eventdev.

Distributed Scheduler

The software eventdev is a centralized scheduler, requiring a service core to perform the required event distribution. This is not really a limitation but rather a design decision.

The `RTE_EVENT_DEV_CAP_DISTRIBUTED_SCHED` flag is not set in the `event_dev_cap` field of the `rte_event_dev_info` struct for the software eventdev.

Dequeue Timeout

The eventdev API supports a timeout when dequeuing packets using the `rte_event_dequeue_burst` function. This allows a core to wait for an event to arrive, or until `timeout` number of ticks have passed. Timeout ticks is not supported by the software eventdev for performance reasons.

14.5 OCTEON TX SSOVF Eventdev Driver

The OCTEON TX SSOVF PMD (**librte_pmd_octeontx_ssovf**) provides poll mode eventdev driver support for the inbuilt event device found in the **Cavium OCTEON TX** SoC family as well as their virtual functions (VF) in SR-IOV context.

More information can be found at Cavium, Inc Official Website.

14.5.1 Features

Features of the OCTEON TX SSOVF PMD are:

- 64 Event queues
- 32 Event ports
- HW event scheduler
- Supports 1M flows per event queue
- Flow based event pipelining

- Flow pinning support in flow based event pipelining
- Queue based event pipelining
- Supports ATOMIC, ORDERED, PARALLEL schedule types per flow
- Event scheduling QoS based on event queue priority
- Open system with configurable amount of outstanding events
- HW accelerated dequeue timeout support to enable power management
- SR-IOV VF
- HW managed event timers support through TIMVF, with high precision and time granularity of 1us.
- Up to 64 event timer adapters.

14.5.2 Supported OCTEON TX SoCs

- CN83xx

14.5.3 Prerequisites

See *OCTEON TX Board Support Package* for setup information.

14.5.4 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_PMD_OCTEONTX_SSOVF` (default `y`)

  Toggle compilation of the `librte_pmd_octeontx_ssovf` driver.

Driver Compilation

To compile the OCTEON TX SSOVF PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-thunderx-linux-gcc install
```

14.5.5 Initialization

The OCTEON TX eventdev is exposed as a vdev device which consists of a set of SSO group and work-slot PCIe VF devices. On EAL initialization, SSO PCIe VF devices will be probed and then the vdev device can be created from the application code, or from the EAL command line based on the number of probed/bound SSO PCIe VF device to DPDK by

- Invoking `rte_vdev_init("event_octeontx")` from the application
- Using `--vdev="event_octeontx"` in the EAL options, which will call rte_vdev_init() internally

Example:

```
./your_eventdev_application --vdev="event_octeontx"
```

14.5.6 Selftest

The functionality of OCTEON TX eventdev can be verified using this option, various unit and functional tests are run to verify the sanity. The tests are run once the vdev creation is successfully complete.

```
--vdev="event_octeontx,selftest=1"
```

14.5.7 Enable TIMvf stats

TIMvf stats can be enabled by using this option, by default the stats are disabled.

```
--vdev="event_octeontx,timvf_stats=1"
```

14.5.8 Limitations

Burst mode support

Burst mode is not supported. Dequeue and Enqueue functions accepts only single event at a time.

Rx adapter support

When eth_octeontx is used as Rx adapter event schedule type `RTE_SCHED_TYPE_PARALLEL` is not supported.

Event timer adapter support

When timvf is used as Event timer adapter the clock source mapping is as follows:

```
RTE_EVENT_TIMER_ADAPTER_CPU_CLK  = TIM_CLK_SRC_SCLK
RTE_EVENT_TIMER_ADAPTER_EXT_CLK0 = TIM_CLK_SRC_GPIO
RTE_EVENT_TIMER_ADAPTER_EXT_CLK1 = TIM_CLK_SRC_GTI
RTE_EVENT_TIMER_ADAPTER_EXT_CLK2 = TIM_CLK_SRC_PTP
```

When timvf is used as Event timer adapter event schedule type `RTE_SCHED_TYPE_PARALLEL` is not supported.

Max mempool size

Max mempool size when using OCTEON TX Eventdev (SSO) should be limited to 128K. When running dpdk-test-eventdev on OCTEON TX the application can limit the number of mbufs by using the option `--pool_sz 131072`

14.6 OCTEON TX2 SSO Eventdev Driver

The OCTEON TX2 SSO PMD (**librte_pmd_octeontx2_event**) provides poll mode eventdev driver support for the inbuilt event device found in the **Marvell OCTEON TX2** SoC family.

More information about OCTEON TX2 SoC can be found at Marvell Official Website.

14.6.1 Features

Features of the OCTEON TX2 SSO PMD are:

- 256 Event queues
- 26 (dual) and 52 (single) Event ports
- HW event scheduler
- Supports 1M flows per event queue
- Flow based event pipelining
- Flow pinning support in flow based event pipelining
- Queue based event pipelining
- Supports ATOMIC, ORDERED, PARALLEL schedule types per flow
- Event scheduling QoS based on event queue priority
- Open system with configurable amount of outstanding events limited only by DRAM
- HW accelerated dequeue timeout support to enable power management
- HW managed event timers support through TIM, with high precision and time granularity of 2.5us.
- Up to 256 TIM rings aka event timer adapters.
- Up to 8 rings traversed in parallel.
- HW managed packets enqueued from ethdev to eventdev exposed through event eth RX adapter.
- N:1 ethernet device Rx queue to Event queue mapping.
- Lockfree Tx from event eth Tx adapter using `DEV_TX_OFFLOAD_MT_LOCKFREE` capability while maintaining receive packet order.
- Full Rx/Tx offload support defined through ethdev queue config.

14.6.2 Prerequisites and Compilation procedure

See *Marvell OCTEON TX2 Platform Guide* for setup information.

14.6.3 Pre-Installation Configuration

Compile time Config Options

The following option can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_PMD_OCTEONTX2_EVENTDEV` (default `y`)

  Toggle compilation of the `librte_pmd_octeontx2_event` driver.

Runtime Config Options

- `Maximum number of in-flight events` (default `8192`)

  In **Marvell OCTEON TX2** the max number of in-flight events are only limited by DRAM size, the `xae_cnt` devargs parameter is introduced to provide upper limit for in-flight events. For example:

```
-w 0002:0e:00.0,xae_cnt=16384
```

- `Force legacy mode`

  The `single_ws` devargs parameter is introduced to force legacy mode i.e single workslot mode in SSO and disable the default dual workslot mode. For example:

```
-w 0002:0e:00.0,single_ws=1
```

- `Event Group QoS support`

  SSO GGRPs i.e. queue uses DRAM & SRAM buffers to hold in-flight events. By default the buffers are assigned to the SSO GGRPs to satisfy minimum HW requirements. SSO is free to assign the remaining buffers to GGRPs based on a preconfigured threshold. We can control the QoS of SSO GGRP by modifying the above mentioned thresholds. GGRPs that have higher importance can be assigned higher thresholds than the rest. The dictionary format is as follows [Qx-XAQ-TAQ-IAQ][Qz-XAQ-TAQ-IAQ] expressed in percentages, 0 represents default. For example:

```
-w 0002:0e:00.0,qos=[1-50-50-50]
```

- `Selftest`

  The functionality of OCTEON TX2 eventdev can be verified using this option, various unit and functional tests are run to verify the sanity. The tests are run once the vdev creation is successfully complete. For example:

```
-w 0002:0e:00.0,selftest=1
```

- `TIM disable NPA`

  By default chunks are allocated from NPA then TIM can automatically free them when traversing the list of chunks. The `tim_disable_npa` devargs parameter disables NPA and uses software mempool to manage chunks For example:

```
-w 0002:0e:00.0,tim_disable_npa=1
```

- TIM modify chunk slots

  The tim_chnk_slots devargs can be used to modify number of chunk slots. Chunks are used to store event timers, a chunk can be visualised as an array where the last element points to the next chunk and rest of them are used to store events. TIM traverses the list of chunks and enqueues the event timers to SSO. The default value is 255 and the max value is 4095. For example:

```
-w 0002:0e:00.0,tim_chnk_slots=1023
```

- TIM enable arm/cancel statistics

  The tim_stats_ena devargs can be used to enable arm and cancel stats of event timer adapter. For example:

```
-w 0002:0e:00.0,tim_stats_ena=1
```

- TIM limit max rings reserved

  The tim_rings_lmt devargs can be used to limit the max number of TIM rings i.e. event timer adapter reserved on probe. Since, TIM rings are HW resources we can avoid starving other applications by not grabbing all the rings. For example:

```
-w 0002:0e:00.0,tim_rings_lmt=5
```

- TIM ring control internal parameters

  When using multiple TIM rings the tim_ring_ctl devargs can be used to control each TIM rings internal parameters uniquely. The following dict format is expected [ring-chnk_slots-disable_npa-stats_ena]. 0 represents default values. For Example:

```
-w 0002:0e:00.0,tim_ring_ctl=[2-1023-1-0]
```

- Lock NPA contexts in NDC

  Lock NPA aura and pool contexts in NDC cache. The device args take hexadecimal bitmask where each bit represent the corresponding aura/pool id.

  For example:

```
-w 0002:0e:00.0,npa_lock_mask=0xf
```

Debugging Options

Table 14.1: OCTEON TX2 event device debug options

#	Component	EAL log command
1	SSO	–log-level='pmd.event.octeontx2,8'
2	TIM	–log-level='pmd.event.octeontx2.timer,8'

14.7 OPDL Eventdev Poll Mode Driver

The OPDL (Ordered Packet Distribution Library) eventdev is a specificimplementation of the eventdev API. It is particularly suited to packetprocessing workloads that have high throughput and low latency requirements.All packets follow the same path through the device. The order in whichpackets follow is determined by the order in which queues are set up.Events are left on the ring until they are transmitted. As a result packetsdo not go out of order

14.7.1 Features

The OPDL eventdev implements a subset of features of the eventdev API;

Queues

- Atomic
- Ordered (Parallel is supported as parallel is a subset of Ordered)
- Single-Link

Ports

- Load balanced (for Atomic, Ordered, Parallel queues)
- Single Link (for single-link queues)

14.7.2 Configuration and Options

The software eventdev is a vdev device, and as such can be created from the application code, or from the EAL command line:

- Call `rte_vdev_init("event_opdl0")` from the application
- Use `--vdev="event_opdl0"` in the EAL options, which will call rte_vdev_init() internally

Example:

```
./your_eventdev_application --vdev="event_opdl0"
```

Single Port Queue

It is possible to create a Single Port Queue `RTE_EVENT_QUEUE_CFG_SINGLE_LINK`. Packets dequeued from this queue do not need to be re-enqueued (as is the case with an ordered queue). The purpose of this queue is to allow for asynchronous handling of packets in the middle of a pipeline. Ordered queues in the middle of a pipeline cannot delete packets.

Queue Dependencies

As stated the order in which packets travel through queues is static in nature. They go through the queues in the order the queues are setup at initialisation `rte_event_queue_setup()`. For example if an application sets up 3 queues, Q0, Q1, Q2 and has 3 associated ports P0, P1, P2 and P3 then packets must be

- Enqueued onto Q0 (typically through P0), then
- Dequeued from Q0 (typically through P1), then
- Enqueued onto Q1 (also through P1), then
- Dequeued from Q2 (typically through P2), then
- Enqueued onto Q3 (also through P2), then
- Dequeued from Q3 (typically through P3) and then transmitted on the relevant eth port

14.7.3 Limitations

The opdl implementation has a number of limitations. These limitations are due to the static nature of the underlying queues. It is because of this that the implementation can achieve such high throughput and low latency

The following list is a comprehensive outline of the what is supported and the limitations / restrictions imposed by the opdl pmd

- The order in which packets moved between queues is static and fixed (dynamic scheduling is not supported).
- NEW, RELEASE are not explicitly supported. RX (first enqueue) implicitly adds NEW event types, and TX (last dequeue) implicitly does RELEASE event types.
- All packets follow the same path through device queues.
- Flows within queues are NOT supported.
- Event priority is NOT supported.
- Once the device is stopped all inflight events are lost. Applications should clear all inflight events before stopping it.
- Each port can only be associated with one queue.
- Each queue can have multiple ports associated with it.
- Each worker core has to dequeue the maximum burst size for that port.
- For performance, the rte_event flow_id should not be updated once packetis enqueued on RX.

Validation & Statistics

Validation can be turned on through a command line parameter

```
--vdev="event_opdl0,do_validation=1,self_test=1"
```

If validation is turned on every packet (as opposed to just the first in each burst), is validated to have come from the right queue. Statistics are also produced in this mode. The statistics are available through the eventdev xstats API. Statistics are per port as follows:

- claim_pkts_requested
- claim_pkts_granted
- claim_non_empty
- claim_empty
- total_cycles

CHAPTER

FIFTEEN

RAWDEV DRIVERS

The following are a list of raw device PMDs, which can be used from an application through rawdev API.

15.1 NXP DPAA2 CMDIF Driver

The DPAA2 CMDIF is an implementation of the rawdev API, that provides communication between the GPP and AIOP (Firmware). This is achieved via using the DPCI devices exposed by MC for GPP <–> AIOP interaction.

More information can be found at NXP Official Website.

15.1.1 Features

The DPAA2 CMDIF implements following features in the rawdev API;

- Getting the object ID of the device (DPCI) using attributes
- I/O to and from the AIOP device using DPCI

15.1.2 Supported DPAA2 SoCs

- LS2084A/LS2044A
- LS2088A/LS2048A
- LS1088A/LS1048A

15.1.3 Prerequisites

See *NXP QorIQ DPAA2 Board Support Package* for setup information

Currently supported by DPDK:

- NXP SDK **19.09+**.
- MC Firmware version **10.18.0** and higher.
- Supported architectures: **arm64 LE**.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

Note: Some part of fslmc bus code (mc flib - object library) routines are dual licensed (BSD & GPLv2).

15.1.4 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_PMD_DPAA2_CMDIF_RAWDEV` (default `y`)

  Toggle compilation of the `lrte_pmd_dpaa2_cmdif` driver.

15.1.5 Enabling logs

For enabling logs, use the following EAL parameter:

```
./your_cmdif_application <EAL args> --log-level=pmd.raw.dpaa2.cmdif,<level>
```

Using `pmd.raw.dpaa2.cmdif` as log matching criteria, all Event PMD logs can be enabled which are lower than logging `level`.

Driver Compilation

To compile the DPAA2 CMDIF PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-dpaa-linux-gcc install
```

15.1.6 Initialization

The DPAA2 CMDIF is exposed as a vdev device which consists of dpci devices. On EAL initialization, dpci devices will be probed and then vdev device can be created from the application code by

- Invoking `rte_vdev_init("dpaa2_dpci")` from the application
- Using `--vdev="dpaa2_dpci"` in the EAL options, which will call rte_vdev_init() internally

Example:

```
./your_cmdif_application <EAL args> --vdev="dpaa2_dpci"
```

Platform Requirement

DPAA2 drivers for DPDK can only work on NXP SoCs as listed in the `Supported DPAA2 SoCs`.

15.2 NXP DPAA2 QDMA Driver

The DPAA2 QDMA is an implementation of the rawdev API, that provide means to initiate a DMA transaction from CPU. The initiated DMA is performed without CPU being involved in the actual DMA transaction. This is achieved via using the DPDMAI device exposed by MC.

More information can be found at NXP Official Website.

15.2.1 Features

The DPAA2 QDMA implements following features in the rawdev API;

- Supports issuing DMA of data within memory without hogging CPU while performing DMA operation.
- Supports configuring to optionally get status of the DMA translation on per DMA operation basis.

15.2.2 Supported DPAA2 SoCs

- LX2160A
- LS2084A/LS2044A
- LS2088A/LS2048A
- LS1088A/LS1048A

15.2.3 Prerequisites

See *NXP QorIQ DPAA2 Board Support Package* for setup information

Currently supported by DPDK:

- NXP SDK **19.09+**.
- MC Firmware version **10.18.0** and higher.
- Supported architectures: **arm64 LE**.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

Note: Some part of fslmc bus code (mc flib - object library) routines are dual licensed (BSD & GPLv2).

15.2.4 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_PMD_DPAA2_QDMA_RAWDEV` (default `y`)

  Toggle compilation of the `lrte_pmd_dpaa2_qdma` driver.

15.2.5 Enabling logs

For enabling logs, use the following EAL parameter:

```
./your_qdma_application <EAL args> --log-level=pmd.raw.dpaa2.qdma,<level>
```

Using `pmd.raw.dpaa2.qdma` as log matching criteria, all Event PMD logs can be enabled which are lower than logging `level`.

Driver Compilation

To compile the DPAA2 QDMA PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-dpaa-linux-gcc install
```

15.2.6 Initialization

The DPAA2 QDMA is exposed as a vdev device which consists of dpdmai devices. On EAL initialization, dpdmai devices will be probed and populated into the rawdevices. The rawdev ID of the device can be obtained using

- Invoking `rte_rawdev_get_dev_id("dpdmai.x")` from the application where x is the object ID of the DPDMAI object created by MC. Use can use this index for further rawdev function calls.

Platform Requirement

DPAA2 drivers for DPDK can only work on NXP SoCs as listed in the `Supported DPAA2 SoCs`.

15.3 IFPGA Rawdev Driver

FPGA is used more and more widely in Cloud and NFV, one primary reason is that FPGA not only provides ASIC performance but also it's more flexible than ASIC.

FPGA uses Partial Reconfigure (PR) Parts of Bit Stream to achieve its flexibility. That means one FPGA Device Bit Stream is divided into many Parts of Bit Stream(each Part of Bit Stream is defined as AFU-Accelerated Function Unit), and each AFU is a hardware acceleration unit which can be dynamically reloaded respectively.

By PR (Partial Reconfiguration) AFUs, one FPGA resources can be time-shared by different users. FPGA hot upgrade and fault tolerance can be provided easily.

The SW IFPGA Rawdev Driver (**ifpga_rawdev**) provides a Rawdev driver that utilizes Intel FPGA Software Stack OPAE(Open Programmable Acceleration Engine) for FPGA management.

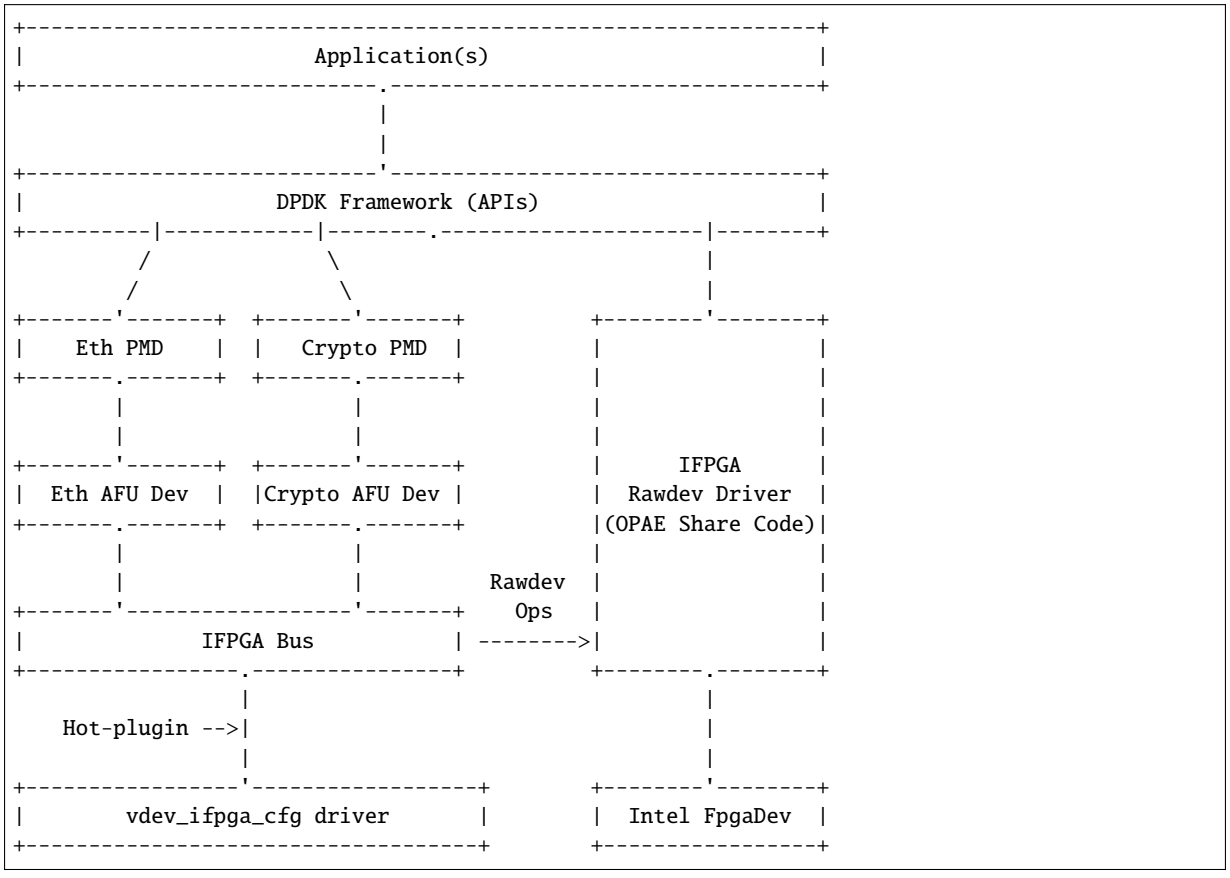
15.3.1 Implementation details

Each instance of IFPGA Rawdev Driver is probed by Intel FpgaDev. In coordination with OPAE share code IFPGA Rawdev Driver provides common FPGA management ops for FPGA operation, OPAE provides all following operations: - FPGA PR (Partial Reconfiguration) management - FPGA AFUs Identifying - FPGA Thermal Management - FPGA Power Management - FPGA Performance reporting - FPGA Remote Debug

All configuration parameters are taken by vdev_ifpga_cfg driver. Besides configuration, vdev_ifpga_cfg driver also hot plugs in IFPGA Bus.

All of the AFUs of one FPGA may share same PCI BDF and AFUs scan depend on IFPGA Rawdev Driver so IFPGA Bus takes AFU device scan and AFU drivers probe. All AFU device driver bind to AFU device by its UUID (Universally Unique Identifier).

To avoid unnecessary code duplication and ensure maximum performance, handling of AFU devices is left to different PMDs; all the design as summarized by the following block diagram:



15.3.2 Build options

- `CONFIG_RTE_LIBRTE_IFPGA_BUS` (default `y`)

  Toggle compilation of IFPGA Bus library.

- `CONFIG_RTE_LIBRTE_IFPGA_RAWDEV` (default `y`)

  Toggle compilation of the `ifpga_rawdev` driver.

15.3.3 Run-time parameters

This driver is invoked automatically in systems added with Intel FPGA, but PR and IFPGA Bus scan is triggered by command line using `--vdev 'ifpga_rawdev_cfg` EAL option.

The following device parameters are supported:

- `ifpga` [string]

  Provide a specific Intel FPGA device PCI BDF. Can be provided multiple times for additional instances.

- `port` [int]

  Each FPGA can provide many channels to PR AFU by software, each channels is identified by this parameter.

- `afu_bts` [string]

  If null, the AFU Bit Stream has been PR in FPGA, if not forces PR and identifies AFU Bit Stream file.

15.4 IOAT Rawdev Driver for Intel® QuickData Technology

The `ioat` rawdev driver provides a poll-mode driver (PMD) for Intel® QuickData Technology, part of Intel® I/O Acceleration Technology (Intel I/OAT). This PMD, when used on supported hardware, allows data copies, for example, cloning packet data, to be accelerated by that hardware rather than having to be done by software, freeing up CPU cycles for other tasks.

15.4.1 Hardware Requirements

On Linux, the presence of an Intel® QuickData Technology hardware can be detected by checking the output of the `lspci` command, where the hardware will be often listed as "Crystal Beach DMA" or "CBDMA". For example, on a system with Intel® Xeon® CPU E5-2699 v4 @2.20GHz, lspci shows:

```
# lspci | grep DMA
00:04.0 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal␣
↪Beach DMA Channel 0 (rev 01)
00:04.1 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal␣
↪Beach DMA Channel 1 (rev 01)
00:04.2 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal␣
↪Beach DMA Channel 2 (rev 01)
00:04.3 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal␣
↪Beach DMA Channel 3 (rev 01)
00:04.4 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal␣
```

(continues on next page)

(continued from previous page)

```
↪Beach DMA Channel 4 (rev 01)
00:04.5 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal␣
↪Beach DMA Channel 5 (rev 01)
00:04.6 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal␣
↪Beach DMA Channel 6 (rev 01)
00:04.7 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal␣
↪Beach DMA Channel 7 (rev 01)
```

On a system with Intel® Xeon® Gold 6154 CPU @ 3.00GHz, lspci shows:

```
# lspci | grep DMA
00:04.0 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.1 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.2 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.3 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.4 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.5 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.6 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.7 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
```

15.4.2 Compilation

For builds done with `make`, the driver compilation is enabled by the `CONFIG_RTE_LIBRTE_PMD_IOAT_RAWDEV` build configuration option. This is enabled by default in builds for x86 platforms, and disabled in other configurations.

For builds using `meson` and `ninja`, the driver will be built when the target platform is x86-based.

15.4.3 Device Setup

The Intel® QuickData Technology HW devices will need to be bound to a user-space IO driver for use. The script `dpdk-devbind.py` script included with DPDK can be used to view the state of the devices and to bind them to a suitable DPDK-supported kernel driver. When querying the status of the devices, they will appear under the category of "Misc (rawdev) devices", i.e. the command `dpdk-devbind.py --status-dev misc` can be used to see the state of those devices alone.

Device Probing and Initialization

Once bound to a suitable kernel device driver, the HW devices will be found as part of the PCI scan done at application initialization time. No vdev parameters need to be passed to create or initialize the device.

Once probed successfully, the device will appear as a `rawdev`, that is a "raw device type" inside DPDK, and can be accessed using APIs from the `rte_rawdev` library.

15.4.4 Using IOAT Rawdev Devices

To use the devices from an application, the rawdev API can be used, along with definitions taken from the device-specific header file `rte_ioat_rawdev.h`. This header is needed to get the definition of structure parameters used by some of the rawdev APIs for IOAT rawdev devices, as well as providing key functions for using the device for memory copies.

Getting Device Information

Basic information about each rawdev device can be queried using the `rte_rawdev_info_get()` API. For most applications, this API will be needed to verify that the rawdev in question is of the expected type. For example, the following code snippet can be used to identify an IOAT rawdev device for use by an application:

```
for (i = 0; i < count && !found; i++) {
        struct rte_rawdev_info info = { .dev_private = NULL };
        found = (rte_rawdev_info_get(i, &info) == 0 &&
                        strcmp(info.driver_name,
                                        IOAT_PMD_RAWDEV_NAME_STR) == 0);
}
```

When calling the `rte_rawdev_info_get()` API for an IOAT rawdev device, the `dev_private` field in the `rte_rawdev_info` struct should either be NULL, or else be set to point to a structure of type `rte_ioat_rawdev_config`, in which case the size of the configured device input ring will be returned in that structure.

Device Configuration

Configuring an IOAT rawdev device is done using the `rte_rawdev_configure()` API, which takes the same structure parameters as the, previously referenced, `rte_rawdev_info_get()` API. The main difference is that, because the parameter is used as input rather than output, the `dev_private` structure element cannot be NULL, and must point to a valid `rte_ioat_rawdev_config` structure, containing the ring size to be used by the device. The ring size must be a power of two, between 64 and 4096.

The following code shows how the device is configured in `test_ioat_rawdev.c`:

```
#define IOAT_TEST_RINGSIZE 512
     struct rte_ioat_rawdev_config p = { .ring_size = -1 };
     struct rte_rawdev_info info = { .dev_private = &p };

     /* ... */

     p.ring_size = IOAT_TEST_RINGSIZE;
     if (rte_rawdev_configure(dev_id, &info) != 0) {
             printf("Error with rte_rawdev_configure()\n");
             return -1;
     }
```

Once configured, the device can then be made ready for use by calling the `rte_rawdev_start()` API.

Performing Data Copies

To perform data copies using IOAT rawdev devices, the functions `rte_ioat_enqueue_copy()` and `rte_ioat_do_copies()` should be used. Once copies have been completed, the completion will be reported back when the application calls `rte_ioat_completed_copies()`.

The `rte_ioat_enqueue_copy()` function enqueues a single copy to the device ring for copying at a later point. The parameters to that function include the IOVA addresses of both the source and destination buffers, as well as two "handles" to be returned to the user when the copy is completed. These handles can be arbitrary values, but two are provided so that the library can track handles for both source and destination on behalf of the user, e.g. virtual addresses for the buffers, or mbuf pointers if packet data is being copied.

While the `rte_ioat_enqueue_copy()` function enqueues a copy operation on the device ring, the copy will not actually be performed until after the application calls the `rte_ioat_do_copies()` function. This function informs the device hardware of the elements enqueued on the ring, and the device will begin to process them. It is expected that, for efficiency reasons, a burst of operations will be enqueued to the device via multiple enqueue calls between calls to the `rte_ioat_do_copies()` function.

The following code from `test_ioat_rawdev.c` demonstrates how to enqueue a burst of copies to the device and start the hardware processing of them:

```
struct rte_mbuf *srcs[32], *dsts[32];
unsigned int j;

for (i = 0; i < RTE_DIM(srcs); i++) {
        char *src_data;

        srcs[i] = rte_pktmbuf_alloc(pool);
        dsts[i] = rte_pktmbuf_alloc(pool);
        srcs[i]->data_len = srcs[i]->pkt_len = length;
        dsts[i]->data_len = dsts[i]->pkt_len = length;
        src_data = rte_pktmbuf_mtod(srcs[i], char *);

        for (j = 0; j < length; j++)
                src_data[j] = rand() & 0xFF;

        if (rte_ioat_enqueue_copy(dev_id,
                        srcs[i]->buf_iova + srcs[i]->data_off,
                        dsts[i]->buf_iova + dsts[i]->data_off,
                        length,
                        (uintptr_t)srcs[i],
                        (uintptr_t)dsts[i],
                        0 /* nofence */) != 1) {
                printf("Error with rte_ioat_enqueue_copy for buffer %u\n",
                                i);
                return -1;
        }
}
rte_ioat_do_copies(dev_id);
```

To retrieve information about completed copies, the API `rte_ioat_completed_copies()` should be used. This API will return to the application a set of completion handles passed in when the relevant copies were enqueued.

The following code from `test_ioat_rawdev.c` shows the test code retrieving information about the completed copies and validating the data is correct before freeing the data buffers using the returned handles:

```
if (rte_ioat_completed_copies(dev_id, 64, (void *)completed_src,
                (void *)completed_dst) != RTE_DIM(srcs)) {
        printf("Error with rte_ioat_completed_copies\n");
        return -1;
}
for (i = 0; i < RTE_DIM(srcs); i++) {
        char *src_data, *dst_data;

        if (completed_src[i] != srcs[i]) {
                printf("Error with source pointer %u\n", i);
                return -1;
        }
        if (completed_dst[i] != dsts[i]) {
                printf("Error with dest pointer %u\n", i);
                return -1;
        }

        src_data = rte_pktmbuf_mtod(srcs[i], char *);
        dst_data = rte_pktmbuf_mtod(dsts[i], char *);
        for (j = 0; j < length; j++)
                if (src_data[j] != dst_data[j]) {
                        printf("Error with copy of packet %u, byte %u\n",
                                        i, j);
                        return -1;
                }
        rte_pktmbuf_free(srcs[i]);
        rte_pktmbuf_free(dsts[i]);
}
```

Querying Device Statistics

The statistics from the IOAT rawdev device can be got via the xstats functions in the `rte_rawdev` library, i.e. `rte_rawdev_xstats_names_get()`, `rte_rawdev_xstats_get()` and `rte_rawdev_xstats_by_name_get`. The statistics returned for each device instance are:

- `failed_enqueues`
- `successful_enqueues`
- `copies_started`
- `copies_completed`

15.5 NTB Rawdev Driver

The `ntb` rawdev driver provides a non-transparent bridge between two separate hosts so that they can communicate with each other. Thus, many user cases can benefit from this, such as fault tolerance and visual acceleration.

This PMD allows two hosts to handshake for device start and stop, memory allocation for the peer to access and read/write allocated memory from peer. Also, the PMD allows to use doorbell registers to notify the peer and share some information by using scratchpad registers.

15.5.1 BIOS setting on Intel Skylake

Intel Non-transparent Bridge needs special BIOS setting. Since the PMD only supports Intel Skylake platform, introduce BIOS setting here. The referencce is https://www.intel.com/content/dam/support/us/en/documents/server-products/Intel_Xeon_Processor_Scalable_Family_BIOS_User_Guide.pdf

- Set the needed PCIe port as NTB to NTB mode on both hosts.
- Enable NTB bars and set bar size of bar 23 and bar 45 as 12-29 (2K-512M) on both hosts. Note that bar size on both hosts should be the same.
- Disable split bars for both hosts.
- Set crosslink control override as DSD/USP on one host, USD/DSP on another host.
- Disable PCIe PII SSC (Spread Spectrum Clocking) for both hosts. This is a hardware requirement.

15.5.2 Build Options

- `CONFIG_RTE_LIBRTE_PMD_NTB_RAWDEV` (default `y`)

  Toggle compilation of the `ntb` driver.

15.5.3 Device Setup

The Intel NTB devices need to be bound to a DPDK-supported kernel driver to use, i.e. igb_uio, vfio. The `dpdk-devbind.py` script can be used to show devices status and to bind them to a suitable kernel driver. They will appear under the category of "Misc (rawdev) devices".

15.5.4 Prerequisites

NTB PMD needs kernel PCI driver to support write combining (WC) to get better performance. The difference will be more than 10 times. To enable WC, there are 2 ways.

- Insert igb_uio with `wc_activate=1` flag if use igb_uio driver.

```
insmod igb_uio.ko wc_activate=1
```

- Enable WC for NTB device's Bar 2 and Bar 4 (Mapped memory) manually. The reference is https://www.kernel.org/doc/html/latest/x86/mtrr.html Get bar base address using `lspci -vvv -s ae:00.0 | grep Region`.

```
# lspci -vvv -s ae:00.0 | grep Region
Region 0: Memory at 39bfe0000000 (64-bit, prefetchable) [size=64K]
Region 2: Memory at 39bfa0000000 (64-bit, prefetchable) [size=512M]
Region 4: Memory at 39bfc0000000 (64-bit, prefetchable) [size=512M]
```

Using the following command to enable WC.

```
echo "base=0x39bfa0000000 size=0x20000000 type=write-combining" >> /proc/mtrr
echo "base=0x39bfc0000000 size=0x20000000 type=write-combining" >> /proc/mtrr
```

And the results:

```
# cat /proc/mtrr
reg00: base=0x000000000 (    0MB), size= 2048MB, count=1: write-back
reg01: base=0x07f000000 ( 2032MB), size=   16MB, count=1: uncachable
reg02: base=0x39bfa0000000 (60553728MB), size=  512MB, count=1: write-combining
reg03: base=0x39bfc0000000 (60554240MB), size=  512MB, count=1: write-combining
```

To disable WC for these regions, using the following.

```
echo "disable=2" >> /proc/mtrr
echo "disable=3" >> /proc/mtrr
```

15.5.5 Ring Layout

Since read/write remote system's memory are through PCI bus, remote read is much more expensive than remote write. Thus, the enqueue and dequeue based on ntb ring should avoid remote read. The ring layout for ntb is like the following:

- Ring Format:

```
desc_ring:

   0               16                                                  64
   +---------------------------------------------------------------+
   |                        buffer address                         |
   +---------------+-----------------------------------------------+
   | buffer length |                      resv                     |
   +---------------+-----------------------------------------------+

used_ring:

   0               16              32
   +---------------+---------------+
   | packet length |     flags     |
   +---------------+---------------+
```

- Ring Layout:

```
+------------------------+   +------------------------+
| used_ring              |   | desc_ring              |
| +---+                  |   | +---+                  |
| |   |                  |   | |   |                  |
| +---+      +---------+ |   | +---+                  |
| |   | ---> | buffer | <+---+-|   |                  |
| +---+      +---------+ |   | +---+                  |
| |   |                  |   | |   |                  |
| +---+                  |   | +---+                  |
|  ...                   |   |  ...                   |
|                        |   |                        |
|            +---------+ |   |            +---------+ |
|            | tx_tail | |   |            | rx_tail | |
| System A   +---------+ |   | System B   +---------+ |
+------------------------+   +------------------------+
             <---------traffic---------
```

- Enqueue and Dequeue Based on this ring layout, enqueue reads rx_tail to get how many free buffers and writes used_ring and tx_tail to tell the peer which buffers are filled with data. And dequeue reads tx_tail to get how many packets are arrived, and writes desc_ring and rx_tail to tell the peer

about the new allocated buffers. So in this way, only remote write happens and remote read can be avoid to get better performance.

15.5.6 Limitation

- This PMD only supports Intel Skylake platform.

15.6 OCTEON TX2 DMA Driver

OCTEON TX2 has an internal DMA unit which can be used by applications to initiate DMA transaction internally, from/to host when OCTEON TX2 operates in PCIe End Point mode. The DMA PF function supports 8 VFs corresponding to 8 DMA queues. Each DMA queue was exposed as a VF function when SRIOV enabled.

15.6.1 Features

This DMA PMD supports below 3 modes of memory transfers

1. Internal - OCTEON TX2 DRAM to DRAM without core intervention
2. Inbound - Host DRAM to OCTEON TX2 DRAM without host/OCTEON TX2 cores involvement
3. Outbound - OCTEON TX2 DRAM to Host DRAM without host/OCTEON TX2 cores involvement

15.6.2 Prerequisites and Compilation procedure

See *Marvell OCTEON TX2 Platform Guide* for setup information.

15.6.3 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_PMD_OCTEONTX2_DMA_RAWDEV` (default `y`)

  Toggle compilation of the `lrte_pmd_octeontx2_dma` driver.

15.6.4 Enabling logs

For enabling logs, use the following EAL parameter:

```
./your_dma_application <EAL args> --log-level=pmd.raw.octeontx2.dpi,<level>
```

Using `pmd.raw.octeontx2.dpi` as log matching criteria, all Event PMD logs can be enabled which are lower than logging `level`.

15.6.5 Initialization

The number of DMA VFs (queues) enabled can be controlled by setting sysfs entry, *sriov_numvfs* for the corresponding PF driver.

```
echo <num_vfs> > /sys/bus/pci/drivers/octeontx2-dpi/0000\:05\:00.0/sriov_numvfs
```

Once the required VFs are enabled, to be accessible from DPDK, VFs need to be bound to vfio-pci driver.

15.6.6 Device Setup

The OCTEON TX2 DPI DMA HW devices will need to be bound to a user-space IO driver for use. The script `dpdk-devbind.py` script included with DPDK can be used to view the state of the devices and to bind them to a suitable DPDK-supported kernel driver. When querying the status of the devices, they will appear under the category of "Misc (rawdev) devices", i.e. the command `dpdk-devbind.py --status-dev misc` can be used to see the state of those devices alone.

15.6.7 Device Configuration

Configuring DMA rawdev device is done using the `rte_rawdev_configure()` API, which takes the mempool as parameter. PMD uses this pool to submit DMA commands to HW.

The following code shows how the device is configured

```
struct dpi_rawdev_conf_s conf = {0};
struct rte_rawdev_info rdev_info = {.dev_private = &conf};

conf.chunk_pool = (void *)rte_mempool_create_empty(...);
rte_mempool_set_ops_byname(conf.chunk_pool, rte_mbuf_platform_mempool_ops(), NULL);
rte_mempool_populate_default(conf.chunk_pool);

rte_rawdev_configure(dev_id, (rte_rawdev_obj_t)&rdev_info);
```

15.6.8 Performing Data Transfer

To perform data transfer using OCTEON TX2 DMA rawdev devices use standard `rte_rawdev_enqueue_buffers()` and `rte_rawdev_dequeue_buffers()` APIs.

15.6.9 Self test

On EAL initialization, dma devices will be probed and populated into the raw devices. The rawdev ID of the device can be obtained using

- Invoke `rte_rawdev_get_dev_id("DPI:x")` from the application where x is the VF device's bus id specified in "bus:device.func" format. Use this index for further rawdev function calls.
- This PMD supports driver self test, to test DMA internal mode from test application one can directly calls `rte_rawdev_selftest(rte_rawdev_get_dev_id("DPI:x"))`

15.7 Marvell OCTEON TX2 End Point Rawdev Driver

OCTEON TX2 has an internal SDP unit which provides End Point mode of operation by exposing its IOQs to Host, IOQs are used for packet I/O between Host and OCTEON TX2. Each OCTEON TX2 SDP PF supports a max of 128 VFs and Each VF is associated with a set of IOQ pairs.

15.7.1 Features

This OCTEON TX2 End Point mode PMD supports

1. Packet Input - Host to OCTEON TX2 with direct data instruction mode.
2. Packet Output - OCTEON TX2 to Host with info pointer mode.

Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_PMD_OCTEONTX2_EP_RAWDEV` (default `y`)

  Toggle compilation of the `lrte_pmd_octeontx2_ep` driver.

15.7.2 Initialization

The number of SDP VFs enabled, can be controlled by setting sysfs entry *sriov_numvfs* for the corresponding PF driver.

```
echo <num_vfs> > /sys/bus/pci/drivers/octeontx2-ep/0000\:04\:00.0/sriov_numvfs
```

Once the required VFs are enabled, to be accessible from DPDK, VFs need to be bound to vfio-pci driver.

15.7.3 Device Setup

The OCTEON TX2 SDP End Point VF devices will need to be bound to a user-space IO driver for use. The script `dpdk-devbind.py` script included with DPDK can be used to view the state of the devices and to bind them to a suitable DPDK-supported kernel driver. When querying the status of the devices, they will appear under the category of "Misc (rawdev) devices", i.e. the command `dpdk-devbind.py --status-dev misc` can be used to see the state of those devices alone.

15.7.4 Device Configuration

Configuring SDP EP rawdev device is done using the `rte_rawdev_configure()` API, which takes the mempool as parameter. PMD uses this pool to send/receive packets to/from the HW.

The following code shows how the device is configured

```
struct sdp_rawdev_info config = {0};
struct rte_rawdev_info rdev_info = {.dev_private = &config};
config.enqdeq_mpool = (void *)rte_mempool_create(...);

rte_rawdev_configure(dev_id, (rte_rawdev_obj_t)&rdev_info);
```

15.7.5 Performing Data Transfer

To perform data transfer using SDP VF EP rawdev devices use standard `rte_rawdev_enqueue_buffers()` and `rte_rawdev_dequeue_buffers()` APIs.

15.7.6 Self test

On EAL initialization, SDP VF devices will be probed and populated into the raw devices. The rawdev ID of the device can be obtained using

- Invoke `rte_rawdev_get_dev_id("SDPEP:x")` from the test application where x is the VF device's bus id specified in "bus:device.func"(BDF) format. Use this index for further rawdev function calls.
- The driver's selftest rawdev API can be used to verify the SDP EP mode functional tests which can send/receive the raw data packets to/from the EP device.

CHAPTER

SIXTEEN

MEMPOOL DEVICE DRIVER

The following are a list of mempool PMDs, which can be used from an application through the mempool API.

16.1 OCTEON TX FPAVF Mempool Driver

The OCTEON TX FPAVF PMD (**librte_mempool_octeontx**) is a mempool driver for offload mempool device found in **Cavium OCTEON TX** SoC family.

More information can be found at Cavium, Inc Official Website.

16.1.1 Features

Features of the OCTEON TX FPAVF PMD are:

- 32 SR-IOV Virtual functions
- 32 Pools
- HW mempool manager

16.1.2 Supported OCTEON TX SoCs

- CN83xx

16.1.3 Prerequisites

See :doc: *../platform/octeontx.rst* for setup information.

16.1.4 Pre-Installation Configuration

Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_MBUF_DEFAULT_MEMPOOL_OPS` (set to `octeontx_fpavf`)

  Set default mempool ops to octeontx_fpavf.

- `CONFIG_RTE_LIBRTE_OCTEONTX_MEMPOOL` (default `y`)

  Toggle compilation of the `librte_mempool_octeontx` driver.

Driver Compilation

To compile the OCTEON TX FPAVF MEMPOOL PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-thunderx-linux-gcc
```

16.1.5 Initialization

The OCTEON TX fpavf mempool initialization similar to other mempool drivers like ring. However user need to pass –base-virtaddr as command line input to application example test_mempool.c application.

Example:

```
./build/app/test -c 0xf --base-virtaddr=0x100000000000 \
                    --mbuf-pool-ops-name="octeontx_fpavf"
```

16.2 OCTEON TX2 NPA Mempool Driver

The OCTEON TX2 NPA PMD (**librte_mempool_octeontx2**) provides mempool driver support for the integrated mempool device found in **Marvell OCTEON TX2** SoC family.

More information about OCTEON TX2 SoC can be found at Marvell Official Website.

16.2.1 Features

OCTEON TX2 NPA PMD supports:

- Up to 128 NPA LFs
- 1M Pools per LF
- HW mempool manager
- Ethdev Rx buffer allocation in HW to save CPU cycles in the Rx path.
- Ethdev Tx buffer recycling in HW to save CPU cycles in the Tx path.

16.2.2 Prerequisites and Compilation procedure

See *Marvell OCTEON TX2 Platform Guide* for setup information.

16.2.3 Pre-Installation Configuration

Compile time Config Options

The following option can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_OCTEONTX2_MEMPOOL` (default `y`)

  Toggle compilation of the `librte_mempool_octeontx2` driver.

Runtime Config Options

- `Maximum number of mempools per application` (default `128`)

  The maximum number of mempools per application needs to be configured on HW during mempool driver initialization. HW can support up to 1M mempools, Since each mempool costs set of HW resources, the `max_pools devargs` parameter is being introduced to configure the number of mempools required for the application. For example:

```
-w 0002:02:00.0,max_pools=512
```

  With the above configuration, the driver will set up only 512 mempools for the given application to save HW resources.

Note: Since this configuration is per application, the end user needs to provide `max_pools` parameter to the first PCIe device probed by the given application.

- `Lock NPA contexts in NDC`

  Lock NPA aura and pool contexts in NDC cache. The device args take hexadecimal bitmask where each bit represent the corresponding aura/pool id.

  For example:

```
-w 0002:02:00.0,npa_lock_mask=0xf
```

Debugging Options

Table 16.1: OCTEON TX2 mempool debug options

#	Component	EAL log command
1	NPA	–log-level='pmd.mempool.octeontx2,8'

Standalone mempool device

The `usertools/dpdk-devbind.py` script shall enumerate all the mempool devices available in the system. In order to avoid, the end user to bind the mempool device prior to use ethdev and/or eventdev device, the respective driver configures an NPA LF and attach to the first probed ethdev or eventdev device. In case, if end user need to run mempool as a standalone device (without ethdev or eventdev), end user needs to bind a mempool device using `usertools/dpdk-devbind.py`

Example command to run `mempool_autotest` test with standalone OCTEONTX2 NPA device:

```
echo "mempool_autotest" | build/app/test -c 0xf0 --mbuf-pool-ops-name="octeontx2_npa
↪"
```

CHAPTER

SEVENTEEN

PLATFORM SPECIFIC GUIDES

The following are platform specific guides and setup information.

17.1 Mellanox BlueField Board Support Package

This document has information about steps to setup Mellanox BlueField platform and common offload HW drivers of **Mellanox BlueField** family SoC.

17.1.1 Supported BlueField family SoCs

- BlueField

17.1.2 Supported BlueField Platforms

- BlueField SmartNIC
- BlueField Reference Platforms
- BlueField Controller Card

17.1.3 Common Offload HW Drivers

1. **NIC Driver**

   See *MLX5 poll mode driver* for Mellanox mlx5 NIC driver information.

2. **Cryptodev Driver**

   This is based on the crypto extension support of armv8. See *ARMv8 Crypto Poll Mode Driver* for armv8 crypto driver information.

Note: BlueField has a variant having no armv8 crypto extension support.

17.1.4 Steps To Setup Platform

Toolchains, OS and drivers can be downloaded and installed individually from the Web. But it is recommended to follow instructions at Mellanox BlueField Software Website.

17.1.5 Compile DPDK

DPDK can be compiled either natively on BlueField platforms or cross-compiled on an x86 based platform.

Native Compilation

Refer to *MLX5 poll mode driver* for prerequisites. Either Mellanox OFED/EN or rdma-core library with corresponding kernel drivers is required.

make build

```
make config T=arm64-bluefield-linux-gcc
make -j
```

meson build

```
meson build
ninja -C build
```

Cross Compilation

Refer to *Cross compile DPDK for ARM64* to install the cross toolchain for ARM64. Base on that, additional header files and libraries are required:

- libibverbs
- libmlx5
- libnl-3
- libnl-route-3

Such header files and libraries can be cross-compiled and installed on to the cross toolchain directory like depicted in *Getting the prerequisite library*, but those can also be simply copied from the filesystem of a working BlueField platform. The following script can be run on a BlueField platform in order to create a supplementary tarball for the cross toolchain.

```
mkdir -p aarch64-linux-gnu/libc
pushd $PWD
cd aarch64-linux-gnu/libc

# Copy libraries
mkdir -p lib64
cp -a /lib64/libibverbs* lib64/
```

(continues on next page)

(continued from previous page)

```
cp -a /lib64/libmlx5* lib64/
cp -a /lib64/libnl-3* lib64/
cp -a /lib64/libnl-route-3* lib64/

# Copy header files
mkdir -p usr/include/infiniband
cp -a /usr/include/infiniband/ib_user_ioctl_verbs.h usr/include/infiniband/
cp -a /usr/include/infiniband/mlx5*.h usr/include/infiniband/
cp -a /usr/include/infiniband/tm_types.h usr/include/infiniband/
cp -a /usr/include/infiniband/verbs*.h usr/include/infiniband/

# Create supplementary tarball
popd
tar cf aarch64-linux-gnu-mlx.tar aarch64-linux-gnu/
```

Then, untar the tarball at the cross toolchain directory on the x86 host.

```
cd $(dirname $(which aarch64-linux-gnu-gcc))/..
tar xf aarch64-linux-gnu-mlx.tar
```

make build

```
make config T=arm64-bluefield-linux-gcc
make -j CROSS=aarch64-linux-gnu- CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n
```

meson build

```
meson build --cross-file config/arm/arm64_bluefield_linux_gcc
ninja -C build
```

17.2 NXP QorIQ DPAA Board Support Package

This doc has information about steps to setup QorIQ dpaa based layerscape platform and information about common offload hw block drivers of **NXP QorIQ DPAA** SoC family.

17.2.1 Supported DPAA SoCs

- LS1046A/LS1026A
- LS1043A/LS1023A

More information about SoC can be found at NXP Official Website.

17.2.2 Common Offload HW Block Drivers

1. **Nics Driver**

   See *DPAA Poll Mode Driver* for NXP dpaa nic driver information.

2. **Cryptodev Driver**

   See *NXP DPAA CAAM (DPAA_SEC)* for NXP dpaa cryptodev driver information.

3. **Eventdev Driver**

   See *NXP DPAA Eventdev Driver* for NXP dpaa eventdev driver information.

17.2.3 Steps To Setup Platform

There are four main pre-requisites for executing DPAA PMD on a DPAA compatible board:

1. **ARM 64 Tool Chain**

   For example, the *aarch64* Linaro Toolchain.

2. **Linux Kernel**

   It can be obtained from NXP's Github hosting.

3. **Rootfile system**

   Any *aarch64* supporting filesystem can be used. For example, Ubuntu 16.04 LTS (Xenial) or 18.04 (Bionic) userland which can be obtained from here.

4. **FMC Tool**

   Before any DPDK application can be executed, the Frame Manager Configuration Tool (FMC) need to be executed to set the configurations of the queues. This includes the queue state, RSS and other policies. This tool can be obtained from NXP (Freescale) Public Git Repository.

   This tool needs configuration files which are available in the *DPDK Extra Scripts*, described below for DPDK usages.

As an alternative method, DPAA PMD can also be executed using images provided as part of SDK from NXP. The SDK includes all the above prerequisites necessary to bring up a DPAA board.

The following dependencies are not part of DPDK and must be installed separately:

- **NXP Linux SDK**

  NXP Linux software development kit (SDK) includes support for family of QorIQ® ARM-Architecture-based system on chip (SoC) processors and corresponding boards.

  It includes the Linux board support packages (BSPs) for NXP SoCs, a fully operational tool chain, kernel and board specific modules.

  SDK and related information can be obtained from: NXP QorIQ SDK.

- **DPDK Extra Scripts**

  DPAA based resources can be configured easily with the help of ready scripts as provided in the DPDK Extra repository.

  DPDK Extras Scripts.

Currently supported by DPDK:

- NXP SDK **2.0+** (preferred: LSDK 18.09).
- Supported architectures: **arm64 LE**.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

17.3 NXP QorIQ DPAA2 Board Support Package

This doc has information about steps to setup NXP QorIQ DPAA2 platform and information about common offload hw block drivers of **NXP QorIQ DPAA2** SoC family.

17.3.1 Supported DPAA2 SoCs

- LX2160A
- LS2084A/LS2044A
- LS2088A/LS2048A
- LS1088A/LS1048A

More information about SoC can be found at NXP Official Website.

17.3.2 Common Offload HW Block Drivers

1. **Nics Driver**

   See *DPAA2 Poll Mode Driver* for NXP dpaa2 nic driver information.

2. **Cryptodev Driver**

   See *NXP DPAA2 CAAM (DPAA2_SEC)* for NXP dpaa2 cryptodev driver information.

3. **Eventdev Driver**

   See *NXP DPAA2 Eventdev Driver* for NXP dpaa2 eventdev driver information.

4. **Rawdev AIOP CMDIF Driver**

   See *NXP DPAA2 CMDIF Driver* for NXP dpaa2 AIOP command interface driver information.

5. **Rawdev QDMA Driver**

   See *NXP DPAA2 QDMA Driver* for NXP dpaa2 QDMA driver information.

17.3.3 Steps To Setup Platform

There are four main pre-requisites for executing DPAA2 PMD on a DPAA2 compatible board:

1. **ARM 64 Tool Chain**

   For example, the *aarch64* Linaro Toolchain.

2. **Linux Kernel**

   It can be obtained from NXP's Github hosting.

3. **Rootfile system**

   Any *aarch64* supporting filesystem can be used. For example, Ubuntu 16.04 LTS (Xenial) or 18.04 (Bionic) userland which can be obtained from here.

4. **Resource Scripts**

   DPAA2 based resources can be configured easily with the help of ready scripts as provided in the DPDK Extra repository.

5. **Build Config**

   Use dpaa build configs, they work for both DPAA2 and DPAA platforms.

As an alternative method, DPAA2 PMD can also be executed using images provided as part of SDK from NXP. The SDK includes all the above prerequisites necessary to bring up a DPAA2 board.

The following dependencies are not part of DPDK and must be installed separately:

- **NXP Linux SDK**

  NXP Linux software development kit (SDK) includes support for family of QorIQ® ARM-Architecture-based system on chip (SoC) processors and corresponding boards.

  It includes the Linux board support packages (BSPs) for NXP SoCs, a fully operational tool chain, kernel and board specific modules.

  SDK and related information can be obtained from: NXP QorIQ SDK.

- **DPDK Extra Scripts**

  DPAA2 based resources can be configured easily with the help of ready scripts as provided in the DPDK Extra repository.

  DPDK Extras Scripts.

Currently supported by DPDK:

- NXP SDK **LSDK 19.09++**.
- MC Firmware version **10.18.0** and higher.
- Supported architectures: **arm64 LE**.
- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

17.4 OCTEON TX Board Support Package

This doc has information about steps to setup OCTEON TX platform and information about common offload hw block drivers of **Cavium OCTEON TX** SoC family.

More information about SoC can be found at Cavium, Inc Official Website.

17.4.1 Common Offload HW Block Drivers

1. **Crypto Driver** See *Cavium OCTEON TX Crypto Poll Mode Driver* for octeontx crypto driver information.
2. **Eventdev Driver** See *OCTEON TX SSOVF Eventdev Driver* for octeontx ssovf eventdev driver information.
3. **Mempool Driver** See *OCTEON TX FPAVF Mempool Driver* for octeontx fpavf mempool driver information.

17.4.2 Steps To Setup Platform

There are three main pre-prerequisites for setting up Platform drivers on OCTEON TX compatible board:

1. **OCTEON TX Linux kernel PF driver for Network acceleration HW blocks**

   The OCTEON TX Linux kernel drivers (includes the required PF driver for the Platform drivers) are available on Github at octeontx-kmod along with build, install and dpdk usage instructions.

Note: The PF driver and the required microcode for the crypto offload block will be available with OCTEON TX SDK only. So for using crypto offload, follow the steps mentioned in *Setup Platform Using OCTEON TX SDK*.

---

2. **ARM64 Tool Chain**

   For example, the *aarch64* Linaro Toolchain, which can be obtained from here.

3. **Rootfile system**

   Any *aarch64* supporting filesystem can be used. For example, Ubuntu 15.10 (Wily) or 16.04 LTS (Xenial) userland which can be obtained from http://cdimage.ubuntu.com/ubuntu-base/releases/16.04/release/ubuntu-base-16.04.1-base-arm64.tar.gz.

   As an alternative method, Platform drivers can also be executed using images provided as part of SDK from Cavium. The SDK includes all the above prerequisites necessary to bring up a OCTEON TX board. Please refer *Setup Platform Using OCTEON TX SDK*.

- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

17.4.3 Setup Platform Using OCTEON TX SDK

The OCTEON TX platform drivers can be compiled either natively on **OCTEON TX** ® board or cross-compiled on an x86 based platform.

The **OCTEON TX** ® board must be running the linux kernel based on OCTEON TX SDK 6.2.0 patch 3. In this, the PF drivers for all hardware offload blocks are already built in.

Native Compilation

If the kernel and modules are cross-compiled and copied to the target board, some intermediate binaries required for native build would be missing on the target board. To make sure all the required binaries are available in the native architecture, the linux sources need to be compiled once natively.

```
cd /lib/modules/$(uname -r)/source
make menuconfig
make
```

The above steps would rebuild the modules and the required intermediate binaries. Once the target is ready for native compilation, the OCTEON TX platform drivers can be compiled with the following steps,

```
cd <dpdk directory>
make config T=arm64-thunderx-linux-gcc
make
```

The example applications can be compiled using the following:

```
cd <dpdk directory>
export RTE_SDK=$PWD
export RTE_TARGET=build
cd examples/<application>
make
```

Cross Compilation

The DPDK applications can be cross-compiled on any x86 based platform. The OCTEON TX SDK need to be installed on the build system. The SDK package will provide the required toolchain etc.

Refer to *Cross compile DPDK for ARM64* for further steps on compilation. The 'host' & 'CC' to be used in the commands would change, in addition to the paths to which libnuma related files have to be copied.

The following steps can be used to perform cross-compilation with OCTEON TX SDK 6.2.0 patch 3:

```
cd <sdk_install_dir>
source env-setup

git clone https://github.com/numactl/numactl.git
cd numactl
git checkout v2.0.11 -b v2.0.11
./autogen.sh
autoconf -i
./configure --host=aarch64-thunderx-linux CC=aarch64-thunderx-linux-gnu-gcc --prefix=<numa␣
↪install dir>
make install
```

The above steps will prepare build system with numa additions. Now this build system can be used to build applications for **OCTEON TX** ® platforms.

```
cd <dpdk directory>
export RTE_SDK=$PWD
export RTE_KERNELDIR=$THUNDER_ROOT/linux/kernel/linux
make config T=arm64-thunderx-linux-gcc
make -j CROSS=aarch64-thunderx-linux-gnu- CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n EXTRA_
↪CFLAGS="-isystem <numa_install_dir>/include" EXTRA_LDFLAGS="-L<numa_install_dir>/lib -lnuma"
```

If NUMA support is not required, it can be disabled as explained in *Cross compile DPDK for ARM64*.

Following steps could be used in that case.

```
make config T=arm64-thunderx-linux-gcc
make CROSS=aarch64-thunderx-linux-gnu-
```

SDK and related information can be obtained from: Cavium support site.

17.5 Marvell OCTEON TX2 Platform Guide

This document gives an overview of **Marvell OCTEON TX2** RVU H/W block, packet flow and procedure to build DPDK on OCTEON TX2 platform.

More information about OCTEON TX2 SoC can be found at Marvell Official Website.

17.5.1 Supported OCTEON TX2 SoCs

- CN96xx
- CN93xx

17.5.2 OCTEON TX2 Resource Virtualization Unit architecture

The Fig. 17.1 diagram depicts the RVU architecture and a resource provisioning example.

Fig. 17.1: OCTEON TX2 Resource virtualization architecture and provisioning example

Resource Virtualization Unit (RVU) on Marvell's OCTEON TX2 SoC maps HW resources belonging to the network, crypto and other functional blocks onto PCI-compatible physical and virtual functions.

Each functional block has multiple local functions (LFs) for provisioning to different PCIe devices. RVU supports multiple PCIe SRIOV physical functions (PFs) and virtual functions (VFs).

The Table 17.1 shows the various local functions (LFs) provided by the RVU and its functional mapping to DPDK subsystem.

Table 17.1: RVU managed functional blocks and its mapping to DPDK subsystem

#	LF	DPDK subsystem mapping
1	NIX	rte_ethdev, rte_tm, rte_event_eth_[rt]x_adapter, rte_security
2	NPA	rte_mempool
3	NPC	rte_flow
4	CPT	rte_cryptodev, rte_event_crypto_adapter
5	SSO	rte_eventdev
6	TIM	rte_event_timer_adapter
7	LBK	rte_ethdev
8	DPI	rte_rawdev
9	SDP	rte_ethdev

PF0 is called the administrative / admin function (AF) and has exclusive privileges to provision RVU functional block's LFs to each of the PF/VF.

PF/VFs communicates with AF via a shared memory region (mailbox).Upon receiving requests from PF/VF, AF does resource provisioning and other HW configuration.

AF is always attached to host, but PF/VFs may be used by host kernel itself, or attached to VMs or to userspace applications like DPDK, etc. So, AF has to handle provisioning/configuration requests sent by any device from any domain.

The AF driver does not receive or process any data. It is only a configuration driver used in control path.

The Fig. 17.1 diagram also shows a resource provisioning example where,

1. PFx and PFx-VF0 bound to Linux netdev driver.
2. PFx-VF1 ethdev driver bound to the first DPDK application.
3. PFy ethdev driver, PFy-VF0 ethdev driver, PFz eventdev driver, PFm-VF0 cryptodev driver bound to the second DPDK application.

17.5.3 LBK HW Access

Loopback HW Unit (LBK) receives packets from NIX-RX and sends packets back to NIX-TX. The loopback block has N channels and contains data buffering that is shared across all channels. The LBK HW Unit is abstracted using ethdev subsystem, Where PF0's VFs are exposed as ethdev device and odd-even pairs of VFs are tied together, that is, packets sent on odd VF end up received on even VF and vice versa. This would enable HW accelerated means of communication between two domains where even VF bound to the first domain and odd VF bound to the second domain.

Typical application usage models are,

1. Communication between the Linux kernel and DPDK application.
2. Exception path to Linux kernel from DPDK application as SW `KNI` replacement.
3. Communication between two different DPDK applications.

17.5.4 SDP interface

System DPI Packet Interface unit(SDP) provides PCIe endpoint support for remote host to DMA packets into and out of OCTEON TX2 SoC. SDP interface comes in to live only when OCTEON TX2 SoC is connected in PCIe endpoint mode. It can be used to send/receive packets to/from remote host machine using input/output queue pairs exposed to it. SDP interface receives input packets from remote host from NIX-RX and sends packets to remote host using NIX-TX. Remote host machine need to use corresponding driver (kernel/user mode) to communicate with SDP interface on OCTEON TX2 SoC. SDP supports single PCIe SRIOV physical function(PF) and multiple virtual functions(VF's). Users can bind PF or VF to use SDP interface and it will be enumerated as ethdev ports.

The primary use case for SDP is to enable the smart NIC use case. Typical usage models are,

1. Communication channel between remote host and OCTEON TX2 SoC over PCIe.
2. Transfer packets received from network interface to remote host over PCIe and vice-versa.

17.5.5 OCTEON TX2 packet flow

The Fig. 17.2 diagram depicts the packet flow on OCTEON TX2 SoC in conjunction with use of various HW accelerators.

Fig. 17.2: OCTEON TX2 packet flow in conjunction with use of HW accelerators

17.5.6 HW Offload Drivers

This section lists dataplane H/W block(s) available in OCTEON TX2 SoC.

1. **Ethdev Driver** See *OCTEON TX2 Poll Mode driver* for NIX Ethdev driver information.
2. **Mempool Driver** See *OCTEON TX2 NPA Mempool Driver* for NPA mempool driver information.
3. **Event Device Driver** See *OCTEON TX2 SSO Eventdev Driver* for SSO event device driver information.
4. **DMA Rawdev Driver** See *OCTEON TX2 DMA Driver* for DMA driver information.
5. **Crypto Device Driver** See *Marvell OCTEON TX2 Crypto Poll Mode Driver* for CPT crypto device driver information.

17.5.7 Procedure to Setup Platform

There are three main prerequisites for setting up DPDK on OCTEON TX2 compatible board:

1. **OCTEON TX2 Linux kernel driver**

   The dependent kernel drivers can be obtained from the kernel.org.

   Alternatively, the Marvell SDK also provides the required kernel drivers.

   Linux kernel should be configured with the following features enabled:

```
# 64K pages enabled for better performance
CONFIG_ARM64_64K_PAGES=y
CONFIG_ARM64_VA_BITS_48=y
# huge pages support enabled
CONFIG_HUGETLBFS=y
CONFIG_HUGETLB_PAGE=y
# VFIO enabled with TYPE1 IOMMU at minimum
CONFIG_VFIO_IOMMU_TYPE1=y
CONFIG_VFIO_VIRQFD=y
CONFIG_VFIO=y
CONFIG_VFIO_NOIOMMU=y
CONFIG_VFIO_PCI=y
CONFIG_VFIO_PCI_MMAP=y
# SMMUv3 driver
CONFIG_ARM_SMMU_V3=y
# ARMv8.1 LSE atomics
CONFIG_ARM64_LSE_ATOMICS=y
# OCTEONTX2 drivers
CONFIG_OCTEONTX2_MBOX=y
CONFIG_OCTEONTX2_AF=y
# Enable if netdev PF driver required
CONFIG_OCTEONTX2_PF=y
```

(continues on next page)

(continued from previous page)

```
# Enable if netdev VF driver required
CONFIG_OCTEONTX2_VF=y
CONFIG_CRYPTO_DEV_OCTEONTX2_CPT=y
# Enable if OCTEONTX2 DMA PF driver required
CONFIG_OCTEONTX2_DPI_PF=n
```

2. **ARM64 Linux Tool Chain**

   For example, the *aarch64* Linaro Toolchain, which can be obtained from here.

   Alternatively, the Marvell SDK also provides GNU GCC toolchain, which is optimized for OCTEON TX2 CPU.

3. **Rootfile system**

   Any *aarch64* supporting filesystem may be used. For example, Ubuntu 15.10 (Wily) or 16.04 LTS (Xenial) userland which can be obtained from http://cdimage.ubuntu.com/ubuntu-base/releases/16.04/release/ubuntu-base-16.04.1-base-arm64.tar.gz.

   Alternatively, the Marvell SDK provides the buildroot based root filesystem. The SDK includes all the above prerequisites necessary to bring up the OCTEON TX2 board.

- Follow the DPDK *Getting Started Guide for Linux* to setup the basic DPDK environment.

17.5.8 Debugging Options

Table 17.2: OCTEON TX2 common debug options

#	Component	EAL log command
1	Common	–log-level='pmd.octeontx2.base,8'
2	Mailbox	–log-level='pmd.octeontx2.mbox,8'

Debugfs support

The **OCTEON TX2 Linux kernel driver** provides support to dump RVU blocks context or stats using debugfs.

Enable `debugfs` by:

1. Compile kernel with debugfs enabled, i.e `CONFIG_DEBUGFS=y`.
2. Boot OCTEON TX2 with debugfs supported kernel.
3. Verify `debugfs` mounted by default "mount | grep -i debugfs" or mount it manually by using.

```
# mount -t debugfs none /sys/kernel/debug
```

Currently `debugfs` supports the following RVU blocks NIX, NPA, NPC, NDC, SSO & CGX.

The file structure under `/sys/kernel/debug` is as follows

```
octeontx2/
|-- cgx
|   |-- cgx0
|   |   '-- lmac0
|   |       '-- stats
```

(continues on next page)

(continued from previous page)

```
|   |-- cgx1
|   |   |-- lmac0
|   |   |   '-- stats
|   |   '-- lmac1
|   |       '-- stats
|   '-- cgx2
|       '-- lmac0
|           '-- stats
|-- cpt
|   |-- cpt_engines_info
|   |-- cpt_engines_sts
|   |-- cpt_err_info
|   |-- cpt_lfs_info
|   '-- cpt_pc
|---- nix
|   |-- cq_ctx
|   |-- ndc_rx_cache
|   |-- ndc_rx_hits_miss
|   |-- ndc_tx_cache
|   |-- ndc_tx_hits_miss
|   |-- qsize
|   |-- rq_ctx
|   |-- sq_ctx
|   '-- tx_stall_hwissue
|-- npa
|   |-- aura_ctx
|   |-- ndc_cache
|   |-- ndc_hits_miss
|   |-- pool_ctx
|   '-- qsize
|-- npc
|    |-- mcam_info
|    '-- rx_miss_act_stats
|-- rsrc_alloc
'-- sso
     |-- hws
     |   '-- sso_hws_info
     '-- hwgrp
         |-- sso_hwgrp_aq_thresh
         |-- sso_hwgrp_iaq_walk
         |-- sso_hwgrp_pc
         |-- sso_hwgrp_free_list_walk
         |-- sso_hwgrp_ient_walk
         '-- sso_hwgrp_taq_walk
```

RVU block LF allocation:

```
cat /sys/kernel/debug/octeontx2/rsrc_alloc

pcifunc    NPA    NIX    SSO GROUP    SSOWS    TIM    CPT
PF1         0       0
PF4                 1
PF13                          0, 1     0, 1      0
```

CGX example usage:

```
cat /sys/kernel/debug/octeontx2/cgx/cgx2/lmac0/stats

=======Link Status======
Link is UP 40000 Mbps
```

(continues on next page)

(continued from previous page)

```
=======RX_STATS======
Received packets: 0
Octets of received packets: 0
Received PAUSE packets: 0
Received PAUSE and control packets: 0
Filtered DMAC0 (NIX-bound) packets: 0
Filtered DMAC0 (NIX-bound) octets: 0
Packets dropped due to RX FIFO full: 0
Octets dropped due to RX FIFO full: 0
Error packets: 0
Filtered DMAC1 (NCSI-bound) packets: 0
Filtered DMAC1 (NCSI-bound) octets: 0
NCSI-bound packets dropped: 0
NCSI-bound octets dropped: 0
=======TX_STATS======
Packets dropped due to excessive collisions: 0
Packets dropped due to excessive deferral: 0
Multiple collisions before successful transmission: 0
Single collisions before successful transmission: 0
Total octets sent on the interface: 0
Total frames sent on the interface: 0
Packets sent with an octet count < 64: 0
Packets sent with an octet count == 64: 0
Packets sent with an octet count of 65127: 0
Packets sent with an octet count of 128-255: 0
Packets sent with an octet count of 256-511: 0
Packets sent with an octet count of 512-1023: 0
Packets sent with an octet count of 1024-1518: 0
Packets sent with an octet count of > 1518: 0
Packets sent to a broadcast DMAC: 0
Packets sent to the multicast DMAC: 0
Transmit underflow and were truncated: 0
Control/PAUSE packets sent: 0
```

CPT example usage:

```
cat /sys/kernel/debug/octeontx2/cpt/cpt_pc

CPT instruction requests   0
CPT instruction latency    0
CPT NCB read requests      0
CPT NCB read latency       0
CPT read requests caused by UC fills   0
CPT active cycles pc       1395642
CPT clock count pc         5579867595493
```

NIX example usage:

```
Usage: echo <nixlf> [cq number/all] > /sys/kernel/debug/octeontx2/nix/cq_ctx
       cat /sys/kernel/debug/octeontx2/nix/cq_ctx
echo 0 0 > /sys/kernel/debug/octeontx2/nix/cq_ctx
cat /sys/kernel/debug/octeontx2/nix/cq_ctx

=====cq_ctx for nixlf:0 and qidx:0 is=====
W0: base                        158ef1a00

W1: wrptr                       0
W1: avg_con                     0
W1: cint_idx                    0
W1: cq_err                      0
```

(continues on next page)

(continued from previous page)

```
W1: qint_idx                    0
W1: bpid                        0
W1: bp_ena                      0

W2: update_time                 31043
W2:avg_level                    255
W2: head                        0
W2:tail                         0

W3: cq_err_int_ena              5
W3:cq_err_int                   0
W3: qsize                       4
W3:caching                      1
W3: substream                   0x000
W3: ena                                 1
W3: drop_ena                    1
W3: drop                        64
W3: bp                          0
```

NPA example usage:

```
Usage: echo <npalf> [pool number/all] > /sys/kernel/debug/octeontx2/npa/pool_ctx
       cat /sys/kernel/debug/octeontx2/npa/pool_ctx
echo 0 0 > /sys/kernel/debug/octeontx2/npa/pool_ctx
cat /sys/kernel/debug/octeontx2/npa/pool_ctx

======POOL : 0=======
W0: Stack base          1375bff00
W1: ena                 1
W1: nat_align           1
W1: stack_caching       1
W1: stack_way_mask      0
W1: buf_offset          1
W1: buf_size            19
W2: stack_max_pages     24315
W2: stack_pages         24314
W3: op_pc               267456
W4: stack_offset        2
W4: shift               5
W4: avg_level           255
W4: avg_con             0
W4: fc_ena              0
W4: fc_stype            0
W4: fc_hyst_bits        0
W4: fc_up_crossing      0
W4: update_time         62993
W5: fc_addr             0
W6: ptr_start           1593adf00
W7: ptr_end             180000000
W8: err_int             0
W8: err_int_ena         7
W8: thresh_int          0
W8: thresh_int_ena      0
W8: thresh_up           0
W8: thresh_qint_idx     0
W8: err_qint_idx        0
```

NPC example usage:

```
cat /sys/kernel/debug/octeontx2/npc/mcam_info
```

(continues on next page)

(continued from previous page)

```
NPC MCAM info:
RX keywidth    : 224bits
TX keywidth    : 224bits

MCAM entries   : 2048
Reserved       : 158
Available      : 1890

MCAM counters  : 512
Reserved       : 1
Available      : 511
```

SSO example usage:

```
Usage: echo [<hws>/all] > /sys/kernel/debug/octeontx2/sso/hws/sso_hws_info
echo 0 > /sys/kernel/debug/octeontx2/sso/hws/sso_hws_info

================================================
SSOW HWS[0] Arbitration State      0x0
SSOW HWS[0] Guest Machine Control  0x0
SSOW HWS[0] SET[0] Group Mask[0] 0xffffffffffffffff
SSOW HWS[0] SET[0] Group Mask[1] 0xffffffffffffffff
SSOW HWS[0] SET[0] Group Mask[2] 0xffffffffffffffff
SSOW HWS[0] SET[0] Group Mask[3] 0xffffffffffffffff
SSOW HWS[0] SET[1] Group Mask[0] 0xffffffffffffffff
SSOW HWS[0] SET[1] Group Mask[1] 0xffffffffffffffff
SSOW HWS[0] SET[1] Group Mask[2] 0xffffffffffffffff
SSOW HWS[0] SET[1] Group Mask[3] 0xffffffffffffffff
================================================
```

17.5.9 Compile DPDK

DPDK may be compiled either natively on OCTEON TX2 platform or cross-compiled on an x86 based platform.

Native Compilation

make build

```
make config T=arm64-octeontx2-linux-gcc
make -j
```

The example applications can be compiled using the following:

```
cd <dpdk directory>
export RTE_SDK=$PWD
export RTE_TARGET=build
cd examples/<application>
make -j
```

meson build

```
meson build
ninja -C build
```

Cross Compilation

Refer to *Cross compile DPDK for ARM64* for generic arm64 details.

make build

```
make config T=arm64-octeontx2-linux-gcc
make -j CROSS=aarch64-marvell-linux-gnu- CONFIG_RTE_KNI_KMOD=n
```

meson build

```
meson build --cross-file config/arm/arm64_octeontx2_linux_gcc
ninja -C build
```

Note: By default, meson cross compilation uses `aarch64-linux-gnu-gcc` toolchain, if Marvell toolchain is available then it can be used by overriding the c, cpp, ar, strip `binaries` attributes to respective Marvell toolchain binaries in `config/arm/arm64_octeontx2_linux_gcc` file.

CHAPTER

EIGHTEEN

CONTRIBUTOR'S GUIDELINES

18.1 DPDK Coding Style

18.1.1 Description

This document specifies the preferred style for source files in the DPDK source tree. It is based on the Linux Kernel coding guidelines and the FreeBSD 7.2 Kernel Developer's Manual (see man style(9)), but was heavily modified for the needs of the DPDK.

18.1.2 General Guidelines

The rules and guidelines given in this document cannot cover every situation, so the following general guidelines should be used as a fallback:

- The code style should be consistent within each individual file.
- In the case of creating new files, the style should be consistent within each file in a given directory or module.
- The primary reason for coding standards is to increase code readability and comprehensibility, therefore always use whatever option will make the code easiest to read.

Line length is recommended to be not more than 80 characters, including comments. [Tab stop size should be assumed to be 8-characters wide].

Note: The above is recommendation, and not a hard limit. However, it is expected that the recommendations should be followed in all but the rarest situations.

18.1.3 C Comment Style

Usual Comments

These comments should be used in normal cases. To document a public API, a doxygen-like format must be used: refer to *Doxygen Guidelines*.

```
/*
 * VERY important single-line comments look like this.
 */
```

(continues on next page)

(continued from previous page)

```
/* Most single-line comments look like this. */

/*
 * Multi-line comments look like this.  Make them real sentences. Fill
 * them so they look like real paragraphs.
 */
```

License Header

Each file should begin with a special comment containing the appropriate copyright and license for the file. Generally this is the BSD License, except for code for Linux Kernel modules. After any copyright header, a blank line should be left before any other contents, e.g. include statements in a C file.

18.1.4 C Preprocessor Directives

Header Includes

In DPDK sources, the include files should be ordered as following:

1. libc includes (system includes first)
2. DPDK EAL includes
3. DPDK misc libraries includes
4. application-specific includes

Include files from the local application directory are included using quotes, while includes from other paths are included using angle brackets: "<>".

Example:

```
#include <stdio.h>
#include <stdlib.h>

#include <rte_eal.h>

#include <rte_ring.h>
#include <rte_mempool.h>

#include "application.h"
```

Header File Guards

Headers should be protected against multiple inclusion with the usual:

```
#ifndef _FILE_H_
#define _FILE_H_

/* Code */

#endif /* _FILE_H_ */
```

Macros

Do not `#define` or declare names except with the standard DPDK prefix: `RTE_`. This is to ensure there are no collisions with definitions in the application itself.

The names of "unsafe" macros (ones that have side effects), and the names of macros for manifest constants, are all in uppercase.

The expansions of expression-like macros are either a single token or have outer parentheses. If a macro is an inline expansion of a function, the function name is all in lowercase and the macro has the same name all in uppercase. If the macro encapsulates a compound statement, enclose it in a do-while loop, so that it can be used safely in if statements. Any final statement-terminating semicolon should be supplied by the macro invocation rather than the macro, to make parsing easier for pretty-printers and editors.

For example:

```
#define MACRO(x, y) do {                                        \
        variable = (x) + (y);                                   \
        (y) += 2;                                               \
} while(0)
```

Note: Wherever possible, enums and inline functions should be preferred to macros, since they provide additional degrees of type-safety and can allow compilers to emit extra warnings about unsafe code.

Conditional Compilation

- When code is conditionally compiled using `#ifdef` or `#if`, a comment may be added following the matching `#endif` or `#else` to permit the reader to easily discern where conditionally compiled code regions end.
- This comment should be used only for (subjectively) long regions, regions greater than 20 lines, or where a series of nested `#ifdef`'s may be confusing to the reader. Exceptions may be made for cases where code is conditionally not compiled for the purposes of lint(1), or other tools, even though the uncompiled region may be small.
- The comment should be separated from the `#endif` or `#else` by a single space.
- For short conditionally compiled regions, a closing comment should not be used.
- The comment for `#endif` should match the expression used in the corresponding `#if` or `#ifdef`.
- The comment for `#else` and `#elif` should match the inverse of the expression(s) used in the preceding `#if` and/or `#elif` statements.
- In the comments, the subexpression `defined(FOO)` is abbreviated as "FOO". For the purposes of comments, `#ifndef FOO` is treated as `#if !defined(FOO)`.

```
#ifdef KTRACE
#include <sys/ktrace.h>
#endif

#ifdef COMPAT_43
/* A large region here, or other conditional code. */
#else /* !COMPAT_43 */
/* Or here. */
```

(continues on next page)

(continued from previous page)

```
#endif /* COMPAT_43 */

#ifndef COMPAT_43
/* Yet another large region here, or other conditional code. */
#else /* COMPAT_43 */
/* Or here. */
#endif /* !COMPAT_43 */
```

Note: Conditional compilation should be used only when absolutely necessary, as it increases the number of target binaries that need to be built and tested.

18.1.5 C Types

Integers

For fixed/minimum-size integer values, the project uses the form uintXX_t (from stdint.h) instead of older BSD-style integer identifiers of the form u_intXX_t.

Enumerations

- Enumeration values are all uppercase.

```
enum enumtype { ONE, TWO } et;
```

- Enum types should be used in preference to macros #defining a set of (sequential) values.
- Enum types should be prefixed with `rte_` and the elements by a suitable prefix [generally starting `RTE_<enum>_` - where <enum> is a shortname for the enum type] to avoid namespace collisions.

Bitfields

The developer should group bitfields that are included in the same integer, as follows:

```
struct grehdr {
  uint16_t rec:3,
       srr:1,
       seq:1,
       key:1,
       routing:1,
       csum:1,
       version:3,
       reserved:4,
       ack:1;
/* ... */
}
```

Variable Declarations

In declarations, do not put any whitespace between asterisks and adjacent tokens, except for tokens that are identifiers related to types. (These identifiers are the names of basic types, type qualifiers, and typedef-names other than the one being declared.) Separate these identifiers from asterisks using a single space.

For example:

```
int *x;          /* no space after asterisk */
int * const x;  /* space after asterisk when using a type qualifier */
```

- All externally-visible variables should have an `rte_` prefix in the name to avoid namespace collisions.
- Do not use uppercase letters - either in the form of ALL_UPPERCASE, or CamelCase - in variable names. Lower-case letters and underscores only.

Structure Declarations

- In general, when declaring variables in new structures, declare them sorted by use, then by size (largest to smallest), and then in alphabetical order. Sorting by use means that commonly used variables are used together and that the structure layout makes logical sense. Ordering by size then ensures that as little padding is added to the structure as possible.
- For existing structures, additions to structures should be added to the end so for backward compatibility reasons.
- Each structure element gets its own line.
- Try to make the structure readable by aligning the member names using spaces as shown below.
- Names following extremely long types, which therefore cannot be easily aligned with the rest, should be separated by a single space.

```
struct foo {
        struct foo      *next;          /* List of active foo. */
        struct mumble   amumble;        /* Comment for mumble. */
        int             bar;            /* Try to align the comments. */
        struct verylongtypename *baz;   /* Won't fit with other members */
};
```

- Major structures should be declared at the top of the file in which they are used, or in separate header files if they are used in multiple source files.
- Use of the structures should be by separate variable declarations and those declarations must be extern if they are declared in a header file.
- Externally visible structure definitions should have the structure name prefixed by `rte_` to avoid namespace collisions.

Note: Uses of `bool` in structures are not preferred as is wastes space and it's also not clear as to what type size the bool is. A preferred use of `bool` is mainly as a return type from functions that return true/false, and maybe local variable functions.

Ref: LKML

Queues

Use queue(3) macros rather than rolling your own lists, whenever possible. Thus, the previous example would be better written:

```
#include <sys/queue.h>

struct foo {
        LIST_ENTRY(foo) link;      /* Use queue macros for foo lists. */
        struct mumble   amumble;   /* Comment for mumble. */
        int             bar;       /* Try to align the comments. */
        struct verylongtypename *baz;   /* Won't fit with other members */
};
LIST_HEAD(, foo) foohead;          /* Head of global foo list. */
```

DPDK also provides an optimized way to store elements in lockless rings. This should be used in all data-path code, when there are several consumer and/or producers to avoid locking for concurrent access.

Typedefs

Avoid using typedefs for structure types.

For example, use:

```
struct my_struct_type {
/* ... */
};

struct my_struct_type my_var;
```

rather than:

```
typedef struct my_struct_type {
/* ... */
} my_struct_type;

my_struct_type my_var
```

Typedefs are problematic because they do not properly hide their underlying type; for example, you need to know if the typedef is the structure itself, as shown above, or a pointer to the structure. In addition, they must be declared exactly once, whereas an incomplete structure type can be mentioned as many times as necessary. Typedefs are difficult to use in stand-alone header files. The header that defines the typedef must be included before the header that uses it, or by the header that uses it (which causes namespace pollution), or there must be a back-door mechanism for obtaining the typedef.

Note that #defines used instead of typedefs also are problematic (since they do not propagate the pointer type correctly due to direct text replacement). For example, `#define pint int *` does not work as expected, while `typedef int *pint` does work. As stated when discussing macros, typedefs should be preferred to macros in cases like this.

When convention requires a typedef; make its name match the struct tag. Avoid typedefs ending in `_t`, except as specified in Standard C or by POSIX.

Note: It is recommended to use typedefs to define function pointer types, for reasons of code readability. This is especially true when the function type is used as a parameter to another function.

For example:

```
/**
 * Definition of a remote launch function.
 */
typedef int (lcore_function_t)(void *);

/* launch a function of lcore_function_t type */
int rte_eal_remote_launch(lcore_function_t *f, void *arg, unsigned slave_id);
```

18.1.6 C Indentation

General

- Indentation is a hard tab, that is, a tab character, not a sequence of spaces,

---

Note: Global whitespace rule in DPDK, use tabs for indentation, spaces for alignment.

- Do not put any spaces before a tab for indentation.
- If you have to wrap a long statement, put the operator at the end of the line, and indent again.
- For control statements (if, while, etc.), continuation it is recommended that the next line be indented by two tabs, rather than one, to prevent confusion as to whether the second line of the control statement forms part of the statement body or not. Alternatively, the line continuation may use additional spaces to line up to an appropriately point on the preceding line, for example, to align to an opening brace.

Note: As with all style guidelines, code should match style already in use in an existing file.

```
while (really_long_variable_name_1 == really_long_variable_name_2 &&
    var3 == var4){  /* confusing to read as */
    x = y + z;      /* control stmt body lines up with second line of */
    a = b + c;      /* control statement itself if single indent used */
}

if (really_long_variable_name_1 == really_long_variable_name_2 &&
        var3 == var4){  /* two tabs used */
    x = y + z;          /* statement body no longer lines up */
    a = b + c;
}

z = a + really + long + statement + that + needs +
        two + lines + gets + indented + on + the +
        second + and + subsequent + lines;
```

- Do not add whitespace at the end of a line.
- Do not add whitespace or a blank line at the end of a file.

Control Statements and Loops

- Include a space after keywords (if, while, for, return, switch).
- Do not use braces ({ and }) for control statements with zero or just a single statement, unless that statement is more than a single line in which case the braces are permitted.

```
for (p = buf; *p != '\0'; ++p)
        ;       /* nothing */
for (;;)
        stmt;
for (;;) {
        z = a + really + long + statement + that + needs +
                two + lines + gets + indented + on + the +
                second + and + subsequent + lines;
}
for (;;) {
        if (cond)
                stmt;
}
if (val != NULL)
        val = realloc(val, newsize);
```

- Parts of a for loop may be left empty.

```
for (; cnt < 15; cnt++) {
        stmt1;
        stmt2;
}
```

- Closing and opening braces go on the same line as the else keyword.
- Braces that are not necessary should be left out.

```
if (test)
        stmt;
else if (bar) {
        stmt;
        stmt;
} else
        stmt;
```

Function Calls

- Do not use spaces after function names.
- Commas should have a space after them.
- No spaces after (or [or preceding the] or) characters.

```
error = function(a1, a2);
if (error != 0)
        exit(error);
```

Operators

- Unary operators do not require spaces, binary operators do.
- Do not use parentheses unless they are required for precedence or unless the statement is confusing without them. However, remember that other people may be more easily confused than you.

Exit

Exits should be 0 on success, or 1 on failure.

```
        exit(0);        /*
                         * Avoid obvious comments such as
                         * "Exit 0 on success."
                         */
}
```

Local Variables

- Variables should be declared at the start of a block of code rather than in the middle. The exception to this is when the variable is `const` in which case the declaration must be at the point of first use/assignment.
- When declaring variables in functions, multiple variables per line are OK. However, if multiple declarations would cause the line to exceed a reasonable line length, begin a new set of declarations on the next line rather than using a line continuation.
- Be careful to not obfuscate the code by initializing variables in the declarations, only the last variable on a line should be initialized. If multiple variables are to be initialized when defined, put one per line.
- Do not use function calls in initializers, except for `const` variables.

```
int i = 0, j = 0, k = 0;  /* bad, too many initializer */

char a = 0;         /* OK, one variable per line with initializer */
char b = 0;

float x, y = 0.0;  /* OK, only last variable has initializer */
```

Casts and sizeof

- Casts and sizeof statements are not followed by a space.
- Always write sizeof statements with parenthesis. The redundant parenthesis rules do not apply to sizeof(var) instances.

18.1.7 C Function Definition, Declaration and Use

Prototypes

- It is recommended (and generally required by the compiler) that all non-static functions are prototyped somewhere.
- Functions local to one source module should be declared static, and should not be prototyped unless absolutely necessary.
- Functions used from other parts of code (external API) must be prototyped in the relevant include file.
- Function prototypes should be listed in a logical order, preferably alphabetical unless there is a compelling reason to use a different ordering.
- Functions that are used locally in more than one module go into a separate header file, for example, "extern.h".
- Do not use the `__P` macro.
- Functions that are part of an external API should be documented using Doxygen-like comments above declarations. See *Doxygen Guidelines* for details.
- Functions that are part of the external API must have an `rte_` prefix on the function name.
- Do not use uppercase letters - either in the form of ALL_UPPERCASE, or CamelCase - in function names. Lower-case letters and underscores only.
- When prototyping functions, associate names with parameter types, for example:

```
void function1(int fd); /* good */
void function2(int);    /* bad */
```

- Short function prototypes should be contained on a single line. Longer prototypes, e.g. those with many parameters, can be split across multiple lines. The second and subsequent lines should be further indented as for line statement continuations as described in the previous section.

```
static char *function1(int _arg, const char *_arg2,
       struct foo *_arg3,
       struct bar *_arg4,
       struct baz *_arg5);
static void usage(void);
```

Note: Unlike function definitions, the function prototypes do not need to place the function return type on a separate line.

Definitions

- The function type should be on a line by itself preceding the function.
- The opening brace of the function body should be on a line by itself.

```
static char *
function(int a1, int a2, float fl, int a4)
{
```

- Do not declare functions inside other functions. ANSI C states that such declarations have file scope regardless of the nesting of the declaration. Hiding file declarations in what appears to be a local scope is undesirable and will elicit complaints from a good compiler.
- Old-style (K&R) function declaration should not be used, use ANSI function declarations instead as shown below.
- Long argument lists should be wrapped as described above in the function prototypes section.

```
/*
 * All major routines should have a comment briefly describing what
 * they do. The comment before the "main" routine should describe
 * what the program does.
 */
int
main(int argc, char *argv[])
{
        char *ep;
        long num;
        int ch;
```

18.1.8 C Statement Style and Conventions

NULL Pointers

- NULL is the preferred null pointer constant. Use NULL instead of `(type *)0` or `(type *)NULL`, except where the compiler does not know the destination type e.g. for variadic args to a function.
- Test pointers against NULL, for example, use:

```
if (p == NULL) /* Good, compare pointer to NULL */

if (!p) /* Bad, using ! on pointer */
```

- Do not use ! for tests unless it is a boolean, for example, use:

```
if (*p == '\0') /* check character against (char)0 */
```

Return Value

- Functions which create objects, or allocate memory, should return pointer types, and NULL on error. The error type should be indicated may setting the variable `rte_errno` appropriately.
- Functions which work on bursts of packets, such as RX-like or TX-like functions, should return the number of packets handled.
- Other functions returning int should generally behave like system calls: returning 0 on success and -1 on error, setting `rte_errno` to indicate the specific type of error.
- Where already standard in a given library, the alternative error approach may be used where the negative value is not -1 but is instead `-errno` if relevant, for example, `-EINVAL`. Note, however, to allow consistency across functions returning integer or pointer types, the previous approach is preferred for any new libraries.
- For functions where no error is possible, the function type should be `void` not `int`.
- Routines returning `void *` should not have their return values cast to any pointer type. (Typecasting can prevent the compiler from warning about missing prototypes as any implicit definition of a function returns int, which, unlike `void *`, needs a typecast to assign to a pointer variable.)

Note: The above rule about not typecasting `void *` applies to malloc, as well as to DPDK functions.

- Values in return statements should not be enclosed in parentheses.

Logging and Errors

In the DPDK environment, use the logging interface provided:

```
/* register log types for this application */
int my_logtype1 = rte_log_register("myapp.log1");
int my_logtype2 = rte_log_register("myapp.log2");

/* set global log level to INFO */
rte_log_set_global_level(RTE_LOG_INFO);

/* only display messages higher than NOTICE for log2 (default
 * is DEBUG) */
rte_log_set_level(my_logtype2, RTE_LOG_NOTICE);

/* enable all PMD logs (whose identifier string starts with "pmd.") */
rte_log_set_level_pattern("pmd.*", RTE_LOG_DEBUG);

/* log in debug level */
rte_log_set_global_level(RTE_LOG_DEBUG);
RTE_LOG(DEBUG, my_logtype1, "this is a debug level message\n");
RTE_LOG(INFO, my_logtype1, "this is a info level message\n");
RTE_LOG(WARNING, my_logtype1, "this is a warning level message\n");
RTE_LOG(WARNING, my_logtype2, "this is a debug level message (not displayed)\n");

/* log in info level */
rte_log_set_global_level(RTE_LOG_INFO);
RTE_LOG(DEBUG, my_logtype1, "debug level message (not displayed)\n");
```

Branch Prediction

- When a test is done in a critical zone (called often or in a data path) the code can use the `likely()` and `unlikely()` macros to indicate the expected, or preferred fast path. They are expanded as a compiler builtin and allow the developer to indicate if the branch is likely to be taken or not. Example:

```
#include <rte_branch_prediction.h>
if (likely(x > 1))
  do_stuff();
```

Note: The use of `likely()` and `unlikely()` should only be done in performance critical paths, and only when there is a clearly preferred path, or a measured performance increase gained from doing so. These macros should be avoided in non-performance-critical code.

Static Variables and Functions

- All functions and variables that are local to a file must be declared as `static` because it can often help the compiler to do some optimizations (such as, inlining the code).
- Functions that should be inlined should to be declared as `static inline` and can be defined in a .c or a .h file.

Note: Static functions defined in a header file must be declared as `static inline` in order to prevent compiler warnings about the function being unused.

Const Attribute

The `const` attribute should be used as often as possible when a variable is read-only.

Inline ASM in C code

The `asm` and `volatile` keywords do not have underscores. The AT&T syntax should be used. Input and output operands should be named to avoid confusion, as shown in the following example:

```
asm volatile("outb %[val], %[port]"
        : :
        [port] "dN" (port),
        [val] "a" (val));
```

Control Statements

- Forever loops are done with for statements, not while statements.
- Elements in a switch statement that cascade should have a FALLTHROUGH comment. For example:

```
switch (ch) {         /* Indent the switch. */
case 'a':             /* Don't indent the case. */
        aflag = 1;    /* Indent case body one tab. */
        /* FALLTHROUGH */
case 'b':
        bflag = 1;
        break;
case '?':
default:
        usage();
        /* NOTREACHED */
}
```

18.1.9 Dynamic Logging

DPDK provides infrastructure to perform logging during runtime. This is very useful for enabling debug output without recompilation. To enable or disable logging of a particular topic, the `--log-level` parameter can be provided to EAL, which will change the log level. DPDK code can register topics, which allows the user to adjust the log verbosity for that specific topic.

In general, the naming scheme is as follows: `type.section.name`

- Type is the type of component, where `lib`, `pmd`, `bus` and `user` are the common options.
- Section refers to a specific area, for example a poll-mode-driver for an ethernet device would use `pmd.net`, while an eventdev PMD uses `pmd.event`.
- The name identifies the individual item that the log applies to. The name section must align with the directory that the PMD code resides. See examples below for clarity.

Examples:

- The virtio network PMD in `drivers/net/virtio` uses `pmd.net.virtio`
- The eventdev software poll mode driver in `drivers/event/sw` uses `pmd.event.sw`
- The octeontx mempool driver in `drivers/mempool/octeontx` uses `pmd.mempool.octeontx`
- The DPDK hash library in `lib/librte_hash` uses `lib.hash`

Specializations

In addition to the above logging topic, any PMD or library can further split logging output by using "specializations". A specialization could be the difference between initialization code, and logs of events that occur at runtime.

An example could be the initialization log messages getting one specialization, while another specialization handles mailbox command logging. Each PMD, library or component can create as many specializations as required.

A specialization looks like this:

- Initialization output: `type.section.name.init`
- PF/VF mailbox output: `type.section.name.mbox`

A real world example is the i40e poll mode driver which exposes two specializations, one for initialization `pmd.net.i40e.init` and the other for the remaining driver logs `pmd.net.i40e.driver`.

Note that specializations have no formatting rules, but please follow a precedent if one exists. In order to see all current log topics and specializations, run the `app/test` binary, and use the `dump_log_types`

18.1.10 Python Code

All Python code should work with Python 2.7+ and 3.2+ and be compliant with PEP8 (Style Guide for Python Code).

The `pep8` tool can be used for testing compliance with the guidelines.

18.1.11 Integrating with the Build System

DPDK supports being built in two different ways:

- using `make` - or more specifically "GNU make", i.e. `gmake` on FreeBSD
- using the tools `meson` and `ninja`

Any new library or driver to be integrated into DPDK should support being built with both systems. While building using `make` is a legacy approach, and most build-system enhancements are being done using `meson` and `ninja` there are no plans at this time to deprecate the legacy `make` build system.

Therefore all new component additions should include both a `Makefile` and a `meson.build` file, and should be added to the component lists in both the `Makefile` and `meson.build` files in the relevant top-level directory: either `lib` directory or a `driver` subdirectory.

Makefile Contents

The `Makefile` for the component should be of the following format, where <name> corresponds to the name of the library in question, e.g. hash, lpm, etc. For drivers, the same format of Makefile is used.

```
# pull in basic DPDK definitions, including whether library is to be
# built or not
include $(RTE_SDK)/mk/rte.vars.mk

# library name
LIB = librte_<name>.a

# any library cflags needed. Generally add "-O3 $(WERROR_FLAGS)"
CFLAGS += -O3
CFLAGS += $(WERROR_FLAGS)

# the symbol version information for the library
EXPORT_MAP := rte_<name>_version.map

# all source filenames are stored in SRCS-y
SRCS-$(CONFIG_RTE_LIBRTE_<NAME>) += rte_<name>.c

# install includes
```

(continues on next page)

(continued from previous page)

```
SYMLINK-$(CONFIG_RTE_LIBRTE_<NAME>)-include += rte_<name>.h

# pull in rules to build the library
include $(RTE_SDK)/mk/rte.lib.mk
```

Meson Build File Contents - Libraries

The `meson.build` file for a new DPDK library should be of the following basic format.

```
sources = files('file1.c', ...)
headers = files('file1.h', ...)
```

This will build based on a number of conventions and assumptions within the DPDK itself, for example, that the library name is the same as the directory name in which the files are stored.

For a library `meson.build` file, there are number of variables which can be set, some mandatory, others optional. The mandatory fields are:

sources
: **Default Value = []**. This variable should list out the files to be compiled up to create the library. Files must be specified using the meson `files()` function.

The optional fields are:

build
: **Default Value = true** Used to optionally compile a library, based on its dependencies or environment. When set to "false" the `reason` value, explained below, should also be set to explain to the user why the component is not being built. A simple example of use would be:

```
if not is_linux
        build = false
        reason = 'only supported on Linux'
endif
```

cflags
: **Default Value = [<-march/-mcpu flags>]**. Used to specify any additional cflags that need to be passed to compile the sources in the library.

deps
: **Default Value = ['eal']**. Used to list the internal library dependencies of the library. It should be assigned to using += rather than overwriting using =. The dependencies should be specified as strings, each one giving the name of a DPDK library, without the `librte_` prefix. Dependencies are handled recursively, so specifying e.g. `mempool`, will automatically also make the library depend upon the mempool library's dependencies too - `ring` and `eal`. For libraries that only depend upon EAL, this variable may be omitted from the `meson.build` file. For example:

```
deps += ['ethdev']
```

ext_deps
: **Default Value = []**. Used to specify external dependencies of this library. They should be returned as dependency objects, as returned from the meson `dependency()` or `find_library()` functions. Before returning these, they should be checked to ensure the dependencies have been found, and, if not, the `build` variable should be set to `false`. For example:

```
my_dep = dependency('libX', required: 'false')
if my_dep.found()
        ext_deps += my_dep
else
        build = false
endif
```

headers
: **Default Value = []**. Used to return the list of header files for the library that should be installed to $PREFIX/include when `ninja install` is run. As with source files, these should be specified using the meson `files()` function.

includes:
: **Default Value = []**. Used to indicate any additional header file paths which should be added to the header search path for other libs depending on this library. EAL uses this so that other libraries building against it can find the headers in subdirectories of the main EAL directory. The base directory of each library is always given in the include path, it does not need to be specified here.

name
: **Default Value = library name derived from the directory name**. If a library's .so or .a file differs from that given in the directory name, the name should be specified using this variable. In practice, since the convention is that for a library called `librte_xyz.so`, the sources are stored in a directory `lib/librte_xyz`, this value should never be needed for new libraries.

Note: The name value also provides the name used to find the function version map file, as part of the build process, so if the directory name and library names differ, the `version.map` file should be named consistently with the library, not the directory

---

objs
: **Default Value = []**. This variable can be used to pass to the library build some pre-built objects that were compiled up as part of another target given in the included library `meson.build` file.

reason
: **Default Value = '<unknown reason>'**. This variable should be used when a library is not to be built i.e. when `build` is set to "false", to specify the reason why a library will not be built. For missing dependencies this should be of the form `'missing dependency, "libname"'`.

use_function_versioning
: **Default Value = false**. Specifies if the library in question has ABI versioned functions. If it has, this value should be set to ensure that the C files are compiled twice with suitable parameters for each of shared or static library builds.

Meson Build File Contents - Drivers

For drivers, the values are largely the same as for libraries. The variables supported are:

build
: As above.

cflags
: As above.

deps
: As above.

ext_deps
As above.

includes
Default Value = <driver directory> Some drivers include a base directory for additional source files and headers, so we have this variable to allow the headers from that base directory to be found when compiling driver sources. Should be appended to using += rather than overwritten using =. The values appended should be meson include objects got using the `include_directories()` function. For example:

```
includes += include_directories('base')
```

name
As above, though note that each driver class can define it's own naming scheme for the resulting `.so` files.

objs
As above, generally used for the contents of the `base` directory.

pkgconfig_extra_libs
Default Value = [] This variable is used to pass additional library link flags through to the DPDK pkgconfig file generated, for example, to track any additional libraries that may need to be linked into the build - especially when using static libraries. Anything added here will be appended to the end of the `pkgconfig --libs` output.

reason
As above.

sources [mandatory]
As above

version
As above

18.2 Design

18.2.1 Environment or Architecture-specific Sources

In DPDK and DPDK applications, some code is specific to an architecture (i686, x86_64) or to an executive environment (freebsd or linux) and so on. As far as is possible, all such instances of architecture or env-specific code should be provided via standard APIs in the EAL.

By convention, a file is common if it is not located in a directory indicating that it is specific. For instance, a file located in a subdir of "x86_64" directory is specific to this architecture. A file located in a subdir of "linux" is specific to this execution environment.

Note: Code in DPDK libraries and applications should be generic. The correct location for architecture or executive environment specific code is in the EAL.

When absolutely necessary, there are several ways to handle specific code:

- Use a `#ifdef` with the CONFIG option in the C code. This can be done when the differences are small and they can be embedded in the same C file:

```
#ifdef RTE_ARCH_I686
toto();
#else
titi();
#endif
```

- Use the CONFIG option in the Makefile. This is done when the differences are more significant. In this case, the code is split into two separate files that are architecture or environment specific. This should only apply inside the EAL library.

Note: As in the linux kernel, the CONFIG_ prefix is not used in C code. This is only needed in Makefiles or shell scripts.

Per Architecture Sources

The following config options can be used:

- CONFIG_RTE_ARCH is a string that contains the name of the architecture.
- CONFIG_RTE_ARCH_I686, CONFIG_RTE_ARCH_X86_64, CONFIG_RTE_ARCH_X86_64_32 or CONFIG_RTE_ARCH_PPC_64 are defined only if we are building for those architectures.

Per Execution Environment Sources

The following config options can be used:

- CONFIG_RTE_EXEC_ENV is a string that contains the name of the executive environment.
- CONFIG_RTE_EXEC_ENV_FREEBSD or CONFIG_RTE_EXEC_ENV_LINUX are defined only if we are building for this execution environment.

18.2.2 Library Statistics

Description

This document describes the guidelines for DPDK library-level statistics counter support. This includes guidelines for turning library statistics on and off and requirements for preventing ABI changes when implementing statistics.

Mechanism to allow the application to turn library statistics on and off

Each library that maintains statistics counters should provide a single build time flag that decides whether the statistics counter collection is enabled or not. This flag should be exposed as a variable within the DPDK configuration file. When this flag is set, all the counters supported by current library are collected for all the instances of every object type provided by the library. When this flag is cleared, none of the counters supported by the current library are collected for any instance of any object type provided by the library:

```
# DPDK file config/common_linux, config/common_freebsd, etc.
CONFIG_RTE_<LIBRARY_NAME>_STATS_COLLECT=y/n
```

The default value for this DPDK configuration file variable (either "yes" or "no") is decided by each library.

Prevention of ABI changes due to library statistics support

The layout of data structures and prototype of functions that are part of the library API should not be affected by whether the collection of statistics counters is turned on or off for the current library. In practical terms, this means that space should always be allocated in the API data structures for statistics counters and the statistics related API functions are always built into the code, regardless of whether the statistics counter collection is turned on or off for the current library.

When the collection of statistics counters for the current library is turned off, the counters retrieved through the statistics related API functions should have a default value of zero.

Motivation to allow the application to turn library statistics on and off

It is highly recommended that each library provides statistics counters to allow an application to monitor the library-level run-time events. Typical counters are: number of packets received/dropped/transmitted, number of buffers allocated/freed, number of occurrences for specific events, etc.

However, the resources consumed for library-level statistics counter collection have to be spent out of the application budget and the counters collected by some libraries might not be relevant to the current application. In order to avoid any unwanted waste of resources and/or performance impacts, the application should decide at build time whether the collection of library-level statistics counters should be turned on or off for each library individually.

Library-level statistics counters can be relevant or not for specific applications:

- For Application A, counters maintained by Library X are always relevant and the application needs to use them to implement certain features, such as traffic accounting, logging, application-level statistics, etc. In this case, the application requires that collection of statistics counters for Library X is always turned on.
- For Application B, counters maintained by Library X are only useful during the application debug stage and are not relevant once debug phase is over. In this case, the application may decide to turn on the collection of Library X statistics counters during the debug phase and at a later stage turn them off.
- For Application C, counters maintained by Library X are not relevant at all. It might be that the application maintains its own set of statistics counters that monitor a different set of run-time events (e.g. number of connection requests, number of active users, etc). It might also be that the application uses multiple libraries (Library X, Library Y, etc) and it is interested in the statistics counters of Library Y, but not in those of Library X. In this case, the application may decide to turn the collection of statistics counters off for Library X and on for Library Y.

The statistics collection consumes a certain amount of CPU resources (cycles, cache bandwidth, memory bandwidth, etc) that depends on:

- Number of libraries used by the current application that have statistics counters collection turned on.
- Number of statistics counters maintained by each library per object type instance (e.g. per port, table, pipeline, thread, etc).
- Number of instances created for each object type supported by each library.

- Complexity of the statistics logic collection for each counter: when only some occurrences of a specific event are valid, additional logic is typically needed to decide whether the current occurrence of the event should be counted or not. For example, in the event of packet reception, when only TCP packets with destination port within a certain range should be recorded, conditional branches are usually required. When processing a burst of packets that have been validated for header integrity, counting the number of bits set in a bitmask might be needed.

18.2.3 PF and VF Considerations

The primary goal of DPDK is to provide a userspace dataplane. Managing VFs from a PF driver is a control plane feature and developers should generally rely on the Linux Kernel for that.

Developers should work with the Linux Kernel community to get the required functionality upstream. PF functionality should only be added to DPDK for testing and prototyping purposes while the kernel work is ongoing. It should also be marked with an "EXPERIMENTAL" tag. If the functionality isn't upstreamable then a case can be made to maintain the PF functionality in DPDK without the EXPERIMENTAL tag.

18.3 ABI Policy

18.3.1 Description

This document details the management policy that ensures the long-term stability of the DPDK ABI and API.

18.3.2 General Guidelines

1. Major ABI versions are declared no more frequently than yearly. Compatibility with the major ABI version is mandatory in subsequent releases until a new major ABI version is declared.
2. Major ABI versions are usually but not always declared aligned with a *LTS release*.
3. The ABI version is managed at a project level in DPDK, and is reflected in all non-experimental *library's soname*.
4. The ABI should be preserved and not changed lightly. ABI changes must follow the outlined *deprecation process*.
5. The addition of symbols is generally not problematic. The modification of symbols is managed with *ABI Versioning*.
6. The removal of symbols is considered an *ABI breakage*, once approved these will form part of the next ABI version.
7. Libraries or APIs marked as *experimental* may change without constraint, as they are not considered part of an ABI version. Experimental libraries have the major ABI version `0`.
8. Updates to the *minimum hardware requirements*, which drop support for hardware which was previously supported, should be treated as an ABI change.

Note: In 2019, the DPDK community stated its intention to move to ABI stable releases, over a number of release cycles. This change begins with maintaining ABI stability through one year of DPDK releases

starting from DPDK 19.11. This policy will be reviewed in 2020, with intention of lengthening the stability period. Additional implementation detail can be found in the *release notes*.

What is an ABI?

An ABI (Application Binary Interface) is the set of runtime interfaces exposed by a library. It is similar to an API (Application Programming Interface) but is the result of compilation. It is also effectively cloned when applications link to dynamic libraries. That is to say when an application is compiled to link against dynamic libraries, it is assumed that the ABI remains constant between the time the application is compiled/linked, and the time that it runs. Therefore, in the case of dynamic linking, it is critical that an ABI is preserved, or (when modified), done in such a way that the application is unable to behave improperly or in an unexpected fashion.

Fig. 18.1: Illustration of DPDK API and ABI.

What is an ABI version?

An ABI version is an instance of a library's ABI at a specific release. Certain releases are considered to be milestone releases, the yearly LTS release for example. The ABI of a milestone release may be declared as a 'major ABI version', where this ABI version is then supported for some number of subsequent releases and is annotated in the library's *soname*.

ABI version support in subsequent releases facilitates application upgrades, by enabling applications built against the milestone release to upgrade to subsequent releases of a library without a rebuild.

More details on major ABI version can be found in the *ABI versioning* guide.

18.3.3 The DPDK ABI policy

A new major ABI version is declared no more frequently than yearly, with declarations usually aligning with a LTS release, e.g. ABI 20 for DPDK 19.11. Compatibility with the major ABI version is then mandatory in subsequent releases until the next major ABI version is declared, e.g. ABI 21 for DPDK 20.11.

At the declaration of a major ABI version, major version numbers encoded in libraries' sonames are bumped to indicate the new version, with the minor version reset to `0`. An example would be `librte_eal.so.20.3` would become `librte_eal.so.21.0`.

The ABI may then change multiple times, without warning, between the last major ABI version increment and the HEAD label of the git tree, with the condition that ABI compatibility with the major ABI version is preserved and therefore sonames do not change.

Minor versions are incremented to indicate the release of a new ABI compatible DPDK release, typically the DPDK quarterly releases. An example of this, might be that `librte_eal.so.20.1` would indicate the first ABI compatible DPDK release, following the declaration of the new major ABI version `20`.

An ABI version is supported in all new releases until the next major ABI version is declared. When changing the major ABI version, the release notes will detail all ABI changes.

Fig. 18.2: Mapping of new ABI versions and ABI version compatibility to DPDK releases.

ABI Changes

The ABI may still change after the declaration of a major ABI version, that is new APIs may be still added or existing APIs may be modified.

Warning: Note that, this policy details the method by which the ABI may be changed, with due regard to preserving compatibility and observing deprecation notices. This process however should not be undertaken lightly, as a general rule ABI stability is extremely important for downstream consumers of DPDK. The API should only be changed for significant reasons, such as performance enhancements. API breakages due to changes such as reorganizing public structure fields for aesthetic or readability purposes should be avoided.

The requirements for changing the ABI are:

1. At least 3 acknowledgments of the need to do so must be made on the dpdk.org mailing list.
    - The acknowledgment of the maintainer of the component is mandatory, or if no maintainer is available for the component, the tree/sub-tree maintainer for that component must acknowledge the ABI change instead.
    - The acknowledgment of three members of the technical board, as delegates of the technical board acknowledging the need for the ABI change, is also mandatory.
    - It is also recommended that acknowledgments from different "areas of interest" be sought for each deprecation, for example: from NIC vendors, CPU vendors, end-users, etc.
2. Backward compatibility with the major ABI version must be maintained through *ABI Versioning*, with *forward-only* compatibility offered for any ABI changes that are indicated to be part of the next ABI version.
    - In situations where backward compatibility is not possible, read the section on *ABI Breakages*.
    - No backward or forward compatibility is offered for API changes marked as `experimental`, as described in the section on *Experimental APIs and Libraries*.
    - In situations in which an `experimental` symbol has been stable for some time. When promoting the symbol to become part of the next ABI version, the maintainer may choose to provide an alias to the `experimental` tag, so as not to break consuming applications.
3. If a newly proposed API functionally replaces an existing one, when the new API becomes non-experimental, then the old one is marked with `__rte_deprecated`.
    - The depreciated API should follow the notification process to be removed, see *Examples of Deprecation Notices*.
    - At the declaration of the next major ABI version, those ABI changes then become a formal part of the new ABI and the requirement to preserve ABI compatibility with the last major ABI version is then dropped.
    - The responsibility for removing redundant ABI compatibility code rests with the original contributor of the ABI changes, failing that, then with the contributor's company and then

finally with the maintainer.

Note: Note that forward-only compatibility is offered for those changes made between major ABI versions. As a library's soname can only describe compatibility with the last major ABI version, until the next major ABI version is declared, these changes therefore cannot be resolved as a runtime dependency through the soname. Therefore any application wishing to make use of these ABI changes can only ensure that its runtime dependencies are met through Operating System package versioning.

Note: Updates to the minimum hardware requirements, which drop support for hardware which was previously supported, should be treated as an ABI change, and follow the relevant deprecation policy procedures as above: 3 acks, technical board approval and announcement at least one release in advance.

ABI Breakages

For those ABI changes that are too significant to reasonably maintain multiple symbol versions, there is an amended process. In these cases, ABIs may be updated without the requirement of backward compatibility being provided. These changes must follow the same process *described above* as non-breaking changes, however with the following additional requirements:

1. ABI breaking changes (including an alternative map file) can be included with deprecation notice, in wrapped way by the `RTE_NEXT_ABI` option, to provide more details about oncoming changes. `RTE_NEXT_ABI` wrapper will be removed at the declaration of the next major ABI version.
2. Once approved, and after the deprecation notice has been observed these changes will form part of the next declared major ABI version.

Examples of ABI Changes

The following are examples of allowable ABI changes occurring between declarations of major ABI versions.

- DPDK 19.11 release defines the function `rte_foo()` ; `rte_foo()` is part of the major ABI version `20`.
- DPDK 20.02 release defines a new function `rte_foo(uint8_t bar)`. This is not a problem as long as the symbol `rte_foo@DPDK20` is preserved through *ABI Versioning*.
  - The new function may be marked with the `__rte_experimental` tag for a number of releases, as described in the section *Experimental*.
  - Once `rte_foo(uint8_t bar)` becomes non-experimental, `rte_foo()` is declared as `__rte_deprecated` and an deprecation notice is provided.
- DPDK 19.11 is not re-released to include `rte_foo(uint8_t bar)`, the new version of `rte_foo` only exists from DPDK 20.02 onwards as described in the *note on forward-only compatibility*.
- DPDK 20.02 release defines the experimental function `__rte_experimental rte_baz()`. This function may or may not exist in the DPDK 20.05 release.
- An application `dPacket` wishes to use `rte_foo(uint8_t bar)`, before the declaration of the DPDK `21` major ABI version. The application can only ensure its runtime dependencies are met by

specifying `DPDK (>= 20.2)` as an explicit package dependency, as the soname can only indicate the supported major ABI version.

- At the release of DPDK 20.11, the function `rte_foo(uint8_t bar)` becomes formally part of then new major ABI version DPDK `21` and `rte_foo()` may be removed.

Examples of Deprecation Notices

The following are some examples of ABI deprecation notices which would be added to the Release Notes:

- The Macro `#RTE_FOO` is deprecated and will be removed with ABI version 21, to be replaced with the inline function `rte_foo()`.
- The function `rte_mbuf_grok()` has been updated to include a new parameter in version 20.2. Backwards compatibility will be maintained for this function until the release of the new DPDK major ABI version 21, in DPDK version 20.11.
- The members of `struct rte_foo` have been reorganized in DPDK 20.02 for performance reasons. Existing binary applications will have backwards compatibility in release 20.02, while newly built binaries will need to reference the new structure variant `struct rte_foo2`. Compatibility will be removed in release 20.11, and all applications will require updating and rebuilding to the new structure at that time, which will be renamed to the original `struct rte_foo`.
- Significant ABI changes are planned for the `librte_dostuff` library. The upcoming release 20.02 will not contain these changes, but release 20.11 will, and no backwards compatibility is planned due to the extensive nature of these changes. Binaries using this library built prior to ABI version 21 will require updating and recompilation.

18.3.4 Experimental

APIs

APIs marked as `experimental` are not considered part of an ABI version and may change without warning at any time. Since changes to APIs are most likely immediately after their introduction, as users begin to take advantage of those new APIs and start finding issues with them, new DPDK APIs will be automatically marked as `experimental` to allow for a period of stabilization before they become part of a tracked ABI version.

Note that marking an API as experimental is a multi step process. To mark an API as experimental, the symbols which are desired to be exported must be placed in an EXPERIMENTAL version block in the corresponding libraries' version map script. Secondly, the corresponding prototypes of those exported functions (in the development header files), must be marked with the `__rte_experimental` tag (see `rte_compat.h`). The DPDK build makefiles perform a check to ensure that the map file and the C code reflect the same list of symbols. This check can be circumvented by defining `ALLOW_EXPERIMENTAL_API` during compilation in the corresponding library Makefile.

In addition to tagging the code with `__rte_experimental`, the doxygen markup must also contain the EXPERIMENTAL string, and the MAINTAINERS file should note the EXPERIMENTAL libraries.

For removing the experimental tag associated with an API, deprecation notice is not required. Though, an API should remain in experimental state for at least one release. Thereafter, the normal process of posting patch for review to mailing list can be followed.

After the experimental tag has been formally removed, a tree/sub-tree maintainer may choose to offer an alias to the experimental tag so as not to break applications using the symbol. The alias is then dropped at the declaration of next major ABI version.

Libraries

Libraries marked as `experimental` are entirely not considered part of an ABI version, and may change without warning at any time. Experimental libraries always have a major ABI version of `0` to indicate they exist outside of *ABI Versioning* , with the minor version incremented with each ABI change to library.

18.4 ABI Versioning

This document details the mechanics of ABI version management in DPDK.

18.4.1 What is a library's soname?

System libraries usually adopt the familiar major and minor version naming convention, where major versions (e.g. `librte_eal 20.x, 21.x`) are presumed to be ABI incompatible with each other and minor versions (e.g. `librte_eal 20.1, 20.2`) are presumed to be ABI compatible. A library's soname. is typically used to provide backward compatibility information about a given library, describing the lowest common denominator ABI supported by the library. The soname or logical name for the library, is typically comprised of the library's name and major version e.g. `librte_eal.so.20`.

During an application's build process, a library's soname is noted as a runtime dependency of the application. This information is then used by the dynamic linker when resolving the applications dependencies at runtime, to load a library supporting the correct ABI version. The library loaded at runtime therefore, may be a minor revision supporting the same major ABI version (e.g. `librte_eal.20.2`), as the library used to link the application (e.g `librte_eal.20.0`).

18.4.2 Major ABI versions

An ABI version change to a given library, especially in core libraries such as `librte_mbuf`, may cause an implicit ripple effect on the ABI of it's consuming libraries, causing ABI breakages. There may however be no explicit reason to bump a dependent library's ABI version, as there may have been no obvious change to the dependent library's API, even though the library's ABI compatibility will have been broken.

This interdependence of DPDK libraries, means that ABI versioning of libraries is more manageable at a project level, with all project libraries sharing a **single ABI version**. In addition, the need to maintain a stable ABI for some number of releases as described in the section *ABI Policy*, means that ABI version increments need to carefully planned and managed at a project level.

Major ABI versions are therefore declared typically aligned with an LTS release and is then supported some number of subsequent releases, shared across all libraries. This means that a single project level ABI version, reflected in all individual library's soname, library filenames and associated version maps persists over multiple releases.

```
$ head ./lib/librte_acl/rte_acl_version.map
DPDK_20 {
       global:
...

$ head ./lib/librte_eal/rte_eal_version.map
DPDK_20 {
       global:
...
```

When an ABI change is made between major ABI versions to a given library, a new section is added to that library's version map describing the impending new ABI version, as described in the section *Examples of ABI Macro use*. The library's soname and filename however do not change, e.g. `libacl.so.20`, as ABI compatibility with the last major ABI version continues to be preserved for that library.

```
$ head ./lib/librte_acl/rte_acl_version.map
DPDK_20 {
       global:
...

DPDK_21 {
       global:

} DPDK_20;
...

$ head ./lib/librte_eal/rte_eal_version.map
DPDK_20 {
       global:
...
```

However when a new ABI version is declared, for example DPDK `21`, old depreciated functions may be safely removed at this point and the entire old major ABI version removed, see the section *Deprecating an entire ABI version* on how this may be done.

```
$ head ./lib/librte_acl/rte_acl_version.map
DPDK_21 {
       global:
...

$ head ./lib/librte_eal/rte_eal_version.map
DPDK_21 {
       global:
...
```

At the same time, the major ABI version is changed atomically across all libraries by incrementing the major version in the ABI_VERSION file. This is done globally for all libraries that declare a stable ABI. For libraries marked as EXPERIMENTAL, their major ABI version is always set to 0.

Minor ABI versions

Each non-LTS release will also increment minor ABI version, to permit multiple DPDK versions being installed alongside each other. Both stable and experimental ABI's are versioned using the global version file that is updated at the start of each release cycle, and are managed at the project level.

18.4.3 Versioning Macros

When a symbol is exported from a library to provide an API, it also provides a calling convention (ABI) that is embodied in its name, return type and arguments. Occasionally that function may need to change to accommodate new functionality or behavior. When that occurs, it is may be required to allow for backward compatibility for a time with older binaries that are dynamically linked to the DPDK.

To support backward compatibility the `rte_function_versioning.h` header file provides macros to use when updating exported functions. These macros are used in conjunction with the `rte_<library>_version.map` file for a given library to allow multiple versions of a symbol to exist in a shared library so that older binaries need not be immediately recompiled.

The macros exported are:

- `VERSION_SYMBOL(b, e, n)`: Creates a symbol version table entry binding versioned symbol `b@DPDK_n` to the internal function `be`.
- `BIND_DEFAULT_SYMBOL(b, e, n)`: Creates a symbol version entry instructing the linker to bind references to symbol `b` to the internal symbol `be`.
- `MAP_STATIC_SYMBOL(f, p)`: Declare the prototype `f`, and map it to the fully qualified function `p`, so that if a symbol becomes versioned, it can still be mapped back to the public symbol name.
- `__vsym`: Annotation to be used in a declaration of the internal symbol `be` to signal that it is being used as an implementation of a particular version of symbol `b`.
- `VERSION_SYMBOL_EXPERIMENTAL(b, e)`: Creates a symbol version table entry binding versioned symbol `b@EXPERIMENTAL` to the internal function `be`. The macro is used when a symbol matures to become part of the stable ABI, to provide an alias to experimental for some time.

Examples of ABI Macro use

Updating a public API

Assume we have a function as follows

```
/*
 * Create an acl context object for apps to
 * manipulate
 */
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param)
{
        ...
}
```

Assume that struct rte_acl_ctx is a private structure, and that a developer wishes to enhance the acl api so that a debugging flag can be enabled on a per-context basis. This requires an addition to the structure (which, being private, is safe), but it also requires modifying the code as follows

```
/*
 * Create an acl context object for apps to
 * manipulate
 */
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param, int debug)
{
        ...
}
```

Note also that, being a public function, the header file prototype must also be changed, as must all the call sites, to reflect the new ABI footprint. We will maintain previous ABI versions that are accessible only to previously compiled binaries.

The addition of a parameter to the function is ABI breaking as the function is public, and existing application may use it in its current form. However, the compatibility macros in DPDK allow a developer to use symbol versioning so that multiple functions can be mapped to the same public symbol based on when an application was linked to it. To see how this is done, we start with the requisite libraries version map file. Initially the version map file for the acl library looks like this

```
DPDK_20 {
     global:

     rte_acl_add_rules;
     rte_acl_build;
     rte_acl_classify;
     rte_acl_classify_alg;
     rte_acl_classify_scalar;
     rte_acl_create;
     rte_acl_dump;
     rte_acl_find_existing;
     rte_acl_free;
     rte_acl_ipv4vlan_add_rules;
     rte_acl_ipv4vlan_build;
     rte_acl_list_dump;
     rte_acl_reset;
     rte_acl_reset_rules;
     rte_acl_set_ctx_classify;

     local: *;
};
```

This file needs to be modified as follows

```
DPDK_20 {
     global:

     rte_acl_add_rules;
     rte_acl_build;
     rte_acl_classify;
     rte_acl_classify_alg;
     rte_acl_classify_scalar;
     rte_acl_create;
     rte_acl_dump;
     rte_acl_find_existing;
     rte_acl_free;
     rte_acl_ipv4vlan_add_rules;
     rte_acl_ipv4vlan_build;
     rte_acl_list_dump;
     rte_acl_reset;
```

(continues on next page)

(continued from previous page)

```
    rte_acl_reset_rules;
    rte_acl_set_ctx_classify;

    local: *;
};

DPDK_21 {
    global:
    rte_acl_create;

} DPDK_20;
```

The addition of the new block tells the linker that a new version node DPDK_21 is available, which contains the symbol rte_acl_create, and inherits the symbols from the DPDK_20 node. This list is directly translated into a list of exported symbols when DPDK is compiled as a shared library.

Next, we need to specify in the code which function maps to the rte_acl_create symbol at which versions. First, at the site of the initial symbol definition, we need to update the function so that it is uniquely named, and not in conflict with the public symbol name

```
-struct rte_acl_ctx *
-rte_acl_create(const struct rte_acl_param *param)
+struct rte_acl_ctx * __vsym
+rte_acl_create_v20(const struct rte_acl_param *param)
{
       size_t sz;
       struct rte_acl_ctx *ctx;
       ...
```

Note that the base name of the symbol was kept intact, as this is conducive to the macros used for versioning symbols and we have annotated the function as `__vsym`, an implementation of a versioned symbol . That is our next step, mapping this new symbol name to the initial symbol name at version node 20. Immediately after the function, we add the VERSION_SYMBOL macro.

```
#include <rte_function_versioning.h>

...
VERSION_SYMBOL(rte_acl_create, _v20, 20);
```

Remembering to also add the rte_function_versioning.h header to the requisite c file where these changes are being made. The macro instructs the linker to create a new symbol `rte_acl_create@DPDK_20`, which matches the symbol created in older builds, but now points to the above newly named function. We have now mapped the original rte_acl_create symbol to the original function (but with a new name).

Please see the section *Enabling versioning macros* to enable this macro in the meson/ninja build. Next, we need to create the new `v21` version of the symbol. We create a new function name, with the `v21` suffix, and implement it appropriately.

```
struct rte_acl_ctx * __vsym
rte_acl_create_v21(const struct rte_acl_param *param, int debug);
{
    struct rte_acl_ctx *ctx = rte_acl_create_v20(param);

    ctx->debug = debug;

    return ctx;
}
```

This code serves as our new API call. Its the same as our old call, but adds the new parameter in place. Next we need to map this function to the new default symbol `rte_acl_create@DPDK_21`. To do this, immediately after the function, we add the BIND_DEFAULT_SYMBOL macro.

```
#include <rte_function_versioning.h>

...
BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 21);
```

The macro instructs the linker to create the new default symbol `rte_acl_create@DPDK_21`, which points to the above newly named function.

We finally modify the prototype of the call in the public header file, such that it contains both versions of the symbol and the public API.

```
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param);

struct rte_acl_ctx * __vsym
rte_acl_create_v20(const struct rte_acl_param *param);

struct rte_acl_ctx * __vsym
rte_acl_create_v21(const struct rte_acl_param *param, int debug);
```

And that's it, on the next shared library rebuild, there will be two versions of rte_acl_create, an old DPDK_20 version, used by previously built applications, and a new DPDK_21 version, used by future built applications.

Note: **Before you leave**, please take care reviewing the sections on *mapping static symbols*, *enabling versioning macros*, and *ABI deprecation*.

Mapping static symbols

Now we've taken what was a public symbol, and duplicated it into two uniquely and differently named symbols. We've then mapped each of those back to the public symbol `rte_acl_create` with different version tags. This only applies to dynamic linking, as static linking has no notion of versioning. That leaves this code in a position of no longer having a symbol simply named `rte_acl_create` and a static build will fail on that missing symbol.

To correct this, we can simply map a function of our choosing back to the public symbol in the static build with the `MAP_STATIC_SYMBOL` macro. Generally the assumption is that the most recent version of the symbol is the one you want to map. So, back in the C file where, immediately after `rte_acl_create_v21` is defined, we add this

```
struct rte_acl_ctx * __vsym
rte_acl_create_v21(const struct rte_acl_param *param, int debug)
{
    ...
}
MAP_STATIC_SYMBOL(struct rte_acl_ctx *rte_acl_create(const struct rte_acl_param *param, int␣
↪debug), rte_acl_create_v21);
```

That tells the compiler that, when building a static library, any calls to the symbol `rte_acl_create` should be linked to `rte_acl_create_v21`

Enabling versioning macros

Finally, we need to indicate to the *meson/ninja build system* to enable versioning macros when building the library or driver. In the libraries or driver where we have added symbol versioning, in the `meson.build` file we add the following

```
use_function_versioning = true
```

at the start of the head of the file. This will indicate to the tool-chain to enable the function version macros when building. There is no corresponding directive required for the `make` build system.

Aliasing experimental symbols

In situations in which an `experimental` symbol has been stable for some time, and it becomes a candidate for promotion to the stable ABI. At this time, when promoting the symbol, maintainer may choose to provide an alias to the `experimental` symbol version, so as not to break consuming applications.

The process to provide an alias to `experimental` is similar to that, of *symbol versioning* described above. Assume we have an experimental function `rte_acl_create` as follows:

```
#include <rte_compat.h>

/*
 * Create an acl context object for apps to
 * manipulate
 */
__rte_experimental
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param)
{
...
}
```

In the map file, experimental symbols are listed as part of the `EXPERIMENTAL` version node.

```
DPDK_20 {
     global:
     ...

     local: *;
};

EXPERIMENTAL {
     global:

     rte_acl_create;
};
```

When we promote the symbol to the stable ABI, we simply strip the `__rte_experimental` annotation from the function and move the symbol from the `EXPERIMENTAL` node, to the node of the next major ABI version as follow.

```
/*
 * Create an acl context object for apps to
 * manipulate
 */
```

(continues on next page)

(continued from previous page)

```
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param)
{
       ...
}
```

We then update the map file, adding the symbol `rte_acl_create` to the `DPDK_21` version node.

```
DPDK_20 {
     global:
     ...

     local: *;
};

DPDK_21 {
     global:

     rte_acl_create;
} DPDK_20;
```

Although there are strictly no guarantees or commitments associated with *experimental symbols*, a maintainer may wish to offer an alias to experimental. The process to add an alias to experimental, is similar to the symbol versioning process. Assuming we have an experimental symbol as before, we now add the symbol to both the `EXPERIMENTAL` and `DPDK_21` version nodes.

```
#include <rte_compat.h>;
#include <rte_function_versioning.h>

/*
 * Create an acl context object for apps to
 * manipulate
 */
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param)
{
...
}

__rte_experimental
struct rte_acl_ctx *
rte_acl_create_e(const struct rte_acl_param *param)
{
   return rte_acl_create(param);
}
VERSION_SYMBOL_EXPERIMENTAL(rte_acl_create, _e);

struct rte_acl_ctx *
rte_acl_create_v21(const struct rte_acl_param *param)
{
   return rte_acl_create(param);
}
BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 21);
```

In the map file, we map the symbol to both the `EXPERIMENTAL` and `DPDK_21` version nodes.

```
DPDK_20 {
     global:
     ...
```

(continues on next page)

(continued from previous page)

```
    local: *;
};

DPDK_21 {
    global:

    rte_acl_create;
} DPDK_20;

EXPERIMENTAL {
    global:

    rte_acl_create;
};
```

Note: Please note, similar to *symbol versioning*, when aliasing to experimental you will also need to take care of *mapping static symbols*.

Deprecating part of a public API

Lets assume that you've done the above updates, and in preparation for the next major ABI version you decide you would like to retire the old version of the function. After having gone through the ABI deprecation announcement process, removal is easy. Start by removing the symbol from the requisite version map file:

```
  DPDK_20 {
       global:

       rte_acl_add_rules;
       rte_acl_build;
       rte_acl_classify;
       rte_acl_classify_alg;
       rte_acl_classify_scalar;
       rte_acl_dump;
-      rte_acl_create
       rte_acl_find_existing;
       rte_acl_free;
       rte_acl_ipv4vlan_add_rules;
       rte_acl_ipv4vlan_build;
       rte_acl_list_dump;
       rte_acl_reset;
       rte_acl_reset_rules;
       rte_acl_set_ctx_classify;

       local: *;
  };

  DPDK_21 {
       global:
       rte_acl_create;
  } DPDK_20;
```

Next remove the corresponding versioned export.

```
-VERSION_SYMBOL(rte_acl_create, _v20, 20);
```

Note that the internal function definition could also be removed, but its used in our example by the newer version `v21`, so we leave it in place and declare it as static. This is a coding style choice.

Deprecating an entire ABI version

While removing a symbol from an ABI may be useful, it is more practical to remove an entire version node at once, as is typically done at the declaration of a major ABI version. If a version node completely specifies an API, then removing part of it, typically makes it incomplete. In those cases it is better to remove the entire node.

To do this, start by modifying the version map file, such that all symbols from the node to be removed are merged into the next node in the map.

In the case of our map above, it would transform to look as follows

```
  DPDK_21 {
       global:

       rte_acl_add_rules;
       rte_acl_build;
       rte_acl_classify;
       rte_acl_classify_alg;
       rte_acl_classify_scalar;
       rte_acl_dump;
       rte_acl_create
       rte_acl_find_existing;
       rte_acl_free;
       rte_acl_ipv4vlan_add_rules;
       rte_acl_ipv4vlan_build;
       rte_acl_list_dump;
       rte_acl_reset;
       rte_acl_reset_rules;
       rte_acl_set_ctx_classify;

       local: *;
};
```

Then any uses of BIND_DEFAULT_SYMBOL that pointed to the old node should be updated to point to the new version node in any header files for all affected symbols.

```
-BIND_DEFAULT_SYMBOL(rte_acl_create, _v20, 20);
+BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 21);
```

Lastly, any VERSION_SYMBOL macros that point to the old version node should be removed, taking care to keep, where need old code in place to support newer versions of the symbol.

18.4.4 Running the ABI Validator

The `devtools` directory in the DPDK source tree contains a utility program, `check-abi.sh`, for validating the DPDK ABI based on the libabigail abidiff utility.

The syntax of the `check-abi.sh` utility is:

```
devtools/check-abi.sh <refdir> <newdir>
```

Where <refdir> specifies the directory housing the reference build of DPDK, and <newdir> specifies the DPDK build directory to check the ABI of.

The ABI compatibility is automatically verified when using a build script from `devtools`, if the variable `DPDK_ABI_REF_VERSION` is set with a tag, as described in *ABI check recommendations*.

18.5 DPDK Documentation Guidelines

This document outlines the guidelines for writing the DPDK Guides and API documentation in RST and Doxygen format.

It also explains the structure of the DPDK documentation and shows how to build the Html and PDF versions of the documents.

18.5.1 Structure of the Documentation

The DPDK source code repository contains input files to build the API documentation and User Guides.

The main directories that contain files related to documentation are shown below:

```
lib
|-- librte_acl
|-- librte_cfgfile
|-- librte_cmdline
|-- librte_eal
|   |-- ...
...
doc
|-- api
+-- guides
    |-- freebsd_gsg
    |-- linux_gsg
    |-- prog_guide
    |-- sample_app_ug
    |-- guidelines
    |-- testpmd_app_ug
    |-- rel_notes
    |-- nics
    |-- ...
```

The API documentation is built from Doxygen comments in the header files. These files are mainly in the `lib/librte_*` directories although some of the Poll Mode Drivers in `drivers/net` are also documented with Doxygen.

The configuration files that are used to control the Doxygen output are in the `doc/api` directory.

The user guides such as *The Programmers Guide* and the *FreeBSD* and *Linux Getting Started* Guides are generated from RST markup text files using the Sphinx Documentation Generator.

These files are included in the `doc/guides/` directory. The output is controlled by the `doc/guides/conf.py` file.

18.5.2 Role of the Documentation

The following items outline the roles of the different parts of the documentation and when they need to be updated or added to by the developer.

- **Release Notes**

  The Release Notes document which features have been added in the current and previous releases of DPDK and highlight any known issues. The Releases Notes also contain notifications of features that will change ABI compatibility in the next release.

  Developers should include updates to the Release Notes with patch sets that relate to any of the following sections:

  - New Features
  - Resolved Issues (see below)
  - Known Issues
  - API Changes
  - ABI Changes
  - Shared Library Versions

  Resolved Issues should only include issues from previous releases that have been resolved in the current release. Issues that are introduced and then fixed within a release cycle do not have to be included here.

  Refer to the Release Notes from the previous DPDK release for the correct format of each section.

- **API documentation**

  The API documentation explains how to use the public DPDK functions. The API index page shows the generated API documentation with related groups of functions.

  The API documentation should be updated via Doxygen comments when new functions are added.

- **Getting Started Guides**

  The Getting Started Guides show how to install and configure DPDK and how to run DPDK based applications on different OSes.

  A Getting Started Guide should be added when DPDK is ported to a new OS.

- **The Programmers Guide**

  The Programmers Guide explains how the API components of DPDK such as the EAL, Memzone, Rings and the Hash Library work. It also explains how some higher level functionality such as Packet Distributor, Packet Framework and KNI work. It also shows the build system and explains how to add applications.

  The Programmers Guide should be expanded when new functionality is added to DPDK.

- **App Guides**

  The app guides document the DPDK applications in the `app` directory such as `testpmd`.

The app guides should be updated if functionality is changed or added.

- **Sample App Guides**

  The sample app guides document the DPDK example applications in the examples directory. Generally they demonstrate a major feature such as L2 or L3 Forwarding, Multi Process or Power Management. They explain the purpose of the sample application, how to run it and step through some of the code to explain the major functionality.

  A new sample application should be accompanied by a new sample app guide. The guide for the Skeleton Forwarding app is a good starting reference.

- **Network Interface Controller Drivers**

  The NIC Drivers document explains the features of the individual Poll Mode Drivers, such as software requirements, configuration and initialization.

  New documentation should be added for new Poll Mode Drivers.

- **Guidelines**

  The guideline documents record community process, expectations and design directions.

  They can be extended, amended or discussed by submitting a patch and getting community approval.

18.5.3 Building the Documentation

Dependencies

The following dependencies must be installed to build the documentation:

- Doxygen.
- Sphinx (also called python-sphinx).
- TexLive (at least TexLive-core and the extra Latex support).
- Inkscape.

Doxygen generates documentation from commented source code. It can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install doxygen

# Red Hat/Fedora.
sudo dnf     -y install doxygen
```

Sphinx is a Python documentation tool for converting RST files to Html or to PDF (via LaTeX). For full support with figure and table captioning the latest version of Sphinx can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install python-pip
sudo pip install --upgrade sphinx
sudo pip install --upgrade sphinx_rtd_theme

# Red Hat/Fedora.
sudo dnf     -y install python-pip
sudo pip install --upgrade sphinx
sudo pip install --upgrade sphinx_rtd_theme
```

For further information on getting started with Sphinx see the Sphinx Getting Started.

Note: To get full support for Figure and Table numbering it is best to install Sphinx 1.3.1 or later.

Inkscape is a vector based graphics program which is used to create SVG images and also to convert SVG images to PDF images. It can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install inkscape

# Red Hat/Fedora.
sudo dnf     -y install inkscape
```

TexLive is an installation package for Tex/LaTeX. It is used to generate the PDF versions of the documentation. The main required packages can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install texlive-latex-extra texlive-lang-greek

# Red Hat/Fedora, selective install.
sudo dnf     -y install texlive-collection-latexextra texlive-greek-fontenc
```

Latexmk is a perl script for running LaTeX for resolving cross references, and it also runs auxiliary programs like bibtex, makeindex if necessary, and dvips. It has also a number of other useful capabilities (see man 1 latexmk).

```
# Ubuntu/Debian.
sudo apt-get -y install latexmk

# Red Hat/Fedora.
sudo dnf     -y install latexmk
```

Build commands

The documentation is built using the standard DPDK build system. Some examples are shown below:

- Generate all the documentation targets:

  ```
  make doc
  ```

- Generate the Doxygen API documentation in Html:

  ```
  make doc-api-html
  ```

- Generate the guides documentation in Html:

  ```
  make doc-guides-html
  ```

- Generate the guides documentation in Pdf:

  ```
  make doc-guides-pdf
  ```

The output of these commands is generated in the `build` directory:

```
build/doc
      |-- html
      |   |-- api
      |   +-- guides
      |
      +-- pdf
          +-- guides
```

Note: Make sure to fix any Sphinx or Doxygen warnings when adding or updating documentation.

The documentation output files can be removed as follows:

```
make doc-clean
```

18.5.4 Document Guidelines

Here are some guidelines in relation to the style of the documentation:

- Document the obvious as well as the obscure since it won't always be obvious to the reader. For example an instruction like "Set up 64 2MB Hugepages" is better when followed by a sample commandline or a link to the appropriate section of the documentation.
- Use American English spellings throughout. This can be checked using the `aspell` utility:

  ```
  aspell --lang=en_US --check doc/guides/sample_app_ug/mydoc.rst
  ```

18.5.5 RST Guidelines

The RST (reStructuredText) format is a plain text markup format that can be converted to Html, PDF or other formats. It is most closely associated with Python but it can be used to document any language. It is used in DPDK to document everything apart from the API.

The Sphinx documentation contains a very useful RST Primer which is a good place to learn the minimal set of syntax required to format a document.

The official reStructuredText website contains the specification for the RST format and also examples of how to use it. However, for most developers the RST Primer is a better resource.

The most common guidelines for writing RST text are detailed in the Documenting Python guidelines. The additional guidelines below reiterate or expand upon those guidelines.

Line Length

- Lines in sentences should be less than 80 characters and wrapped at words. Multiple sentences which are not separated by a blank line are joined automatically into paragraphs.
- Lines in literal blocks **must** be less than 80 characters since they are not wrapped by the document formatters and can exceed the page width in PDF documents.

  Long literal command lines can be shown wrapped with backslashes. For example:

```
testpmd -l 2-3 -n 4 \
        --vdev=virtio_user0,path=/dev/vhost-net,queues=2,queue_size=1024 \
        -- -i --tx-offloads=0x0000002c --enable-lro --txq=2 --rxq=2 \
        --txd=1024 --rxd=1024
```

Whitespace

- Standard RST indentation is 3 spaces. Code can be indented 4 spaces, especially if it is copied from source files.
- No tabs. Convert tabs in embedded code to 4 or 8 spaces.
- No trailing whitespace.
- Add 2 blank lines before each section header.
- Add 1 blank line after each section header.
- Add 1 blank line between each line of a list.

Section Headers

- Section headers should use the following underline formats:

```
Level 1 Heading
===============


Level 2 Heading
---------------


Level 3 Heading
~~~~~~~~~~~~~~~


Level 4 Heading
^^^^^^^^^^^^^^^
```

- Level 4 headings should be used sparingly.
- The underlines should match the length of the text.
- In general, the heading should be less than 80 characters, for conciseness.
- As noted above:
  - Add 2 blank lines before each section header.
  - Add 1 blank line after each section header.

Lists

- Bullet lists should be formatted with a leading * as follows:

```
* Item one.

* Item two is a long line that is wrapped and then indented to match
  the start of the previous line.

* One space character between the bullet and the text is preferred.
```

- Numbered lists can be formatted with a leading number but the preference is to use `#.` which will give automatic numbering. This is more convenient when adding or removing items:

```
#. Item one.

#. Item two is a long line that is wrapped and then indented to match
   the start of the previous line.

#. Item three.
```

- Definition lists can be written with or without a bullet:

```
* Item one.

  Some text about item one.

* Item two.

  Some text about item two.
```

- All lists, and sub-lists, must be separated from the preceding text by a blank line. This is a syntax requirement.
- All list items should be separated by a blank line for readability.

Code and Literal block sections

- Inline text that is required to be rendered with a fixed width font should be enclosed in backquotes like this: ``text``, so that it appears like this: `text`.
- Fixed width, literal blocks of texts should be indented at least 3 spaces and prefixed with `::` like this:

```
Here is some fixed width text::

   0x0001 0x0001 0x00FF 0x00FF
```

- It is also possible to specify an encoding for a literal block using the `.. code-block::` directive so that syntax highlighting can be applied. Examples of supported highlighting are:

```
.. code-block:: console
.. code-block:: c
.. code-block:: python
.. code-block:: diff
.. code-block:: none
```

That can be applied as follows:

```
.. code-block:: c

   #include<stdio.h>

   int main() {

      printf("Hello World\n");

      return 0;
   }
```

Which would be rendered as:

```c
#include<stdio.h>

int main() {

   printf("Hello World\n");

   return 0;
}
```

- The default encoding for a literal block using the simplified `::` directive is `none`.
- Lines in literal blocks must be less than 80 characters since they can exceed the page width when converted to PDF documentation. For long literal lines that exceed that limit try to wrap the text at sensible locations. For example a long command line could be documented like this and still work if copied directly from the docs:

```
build/app/testpmd -l 0-2 -n3 --vdev=net_pcap0,iface=eth0     \
                          --vdev=net_pcap1,iface=eth1     \
                          -- -i --nb-cores=2 --nb-ports=2 \
                             --total-num-mbufs=2048
```

- Long lines that cannot be wrapped, such as application output, should be truncated to be less than 80 characters.

Images

- All images should be in SVG scalar graphics format. They should be true SVG XML files and should not include binary formats embedded in a SVG wrapper.
- The DPDK documentation contains some legacy images in PNG format. These will be converted to SVG in time.
- Inkscape is the recommended graphics editor for creating the images. Use some of the older images in `doc/guides/prog_guide/img/` as a template, for example `mbuf1.svg` or `ring-enqueue1.svg`.
- The SVG images should include a copyright notice, as an XML comment.
- Images in the documentation should be formatted as follows:
  - The image should be preceded by a label in the format `.. _figure_XXXX:` with a leading underscore and where `XXXX` is a unique descriptive name.
  - Images should be included using the `.. figure::` directive and the file type should be set to * (not `.svg`). This allows the format of the image to be changed if required, without updating

the documentation.

  - Images must have a caption as part of the `.. figure::` directive.

- Here is an example of the previous three guidelines:

```
.. _figure_mempool:

.. figure:: img/mempool.*

   A mempool in memory with its associated ring.
```

- Images can then be linked to using the `:numref:` directive:

```
The mempool layout is shown in :numref:`figure_mempool`.
```

  This would be rendered as: *The mempool layout is shown in Fig 6.3*.

  Note: The `:numref:` directive requires Sphinx 1.3.1 or later. With earlier versions it will still be rendered as a link but won't have an automatically generated number.

- The caption of the image can be generated, with a link, using the `:ref:` directive:

```
:ref:`figure_mempool`
```

  This would be rendered as: *A mempool in memory with its associated ring.*

Tables

- RST tables should be used sparingly. They are hard to format and to edit, they are often rendered incorrectly in PDF format, and the same information can usually be shown just as clearly with a definition or bullet list.

- Tables in the documentation should be formatted as follows:

  - The table should be preceded by a label in the format `.. _table_XXXX:` with a leading underscore and where `XXXX` is a unique descriptive name.

  - Tables should be included using the `.. table::` directive and must have a caption.

- Here is an example of the previous two guidelines:

```
.. _table_qos_pipes:

.. table:: Sample configuration for QOS pipes.

   +----------+----------+----------+
   | Header 1 | Header 2 | Header 3 |
   |          |          |          |
   +==========+==========+==========+
   | Text     | Text     | Text     |
   +----------+----------+----------+
   | ...      | ...      | ...      |
   +----------+----------+----------+
```

- Tables can be linked to using the `:numref:` and `:ref:` directives, as shown in the previous section for images. For example:

```
The QOS configuration is shown in :numref:`table_qos_pipes`.
```

- Tables should not include merged cells since they are not supported by the PDF renderer.

Hyperlinks

- Links to external websites can be plain URLs. The following is rendered as https://dpdk.org:

```
https://dpdk.org
```

- They can contain alternative text. The following is rendered as Check out DPDK:

```
`Check out DPDK <https://dpdk.org>`_
```

- An internal link can be generated by placing labels in the document with the format `.. _label_name`.
- The following links to the top of this section: *Hyperlinks*:

```
.. _links:

Hyperlinks
~~~~~~~~~~

* The following links to the top of this section: :ref:`links`:
```

Note: The label must have a leading underscore but the reference to it must omit it. This is a frequent cause of errors and warnings.

- The use of a label is preferred since it works across files and will still work if the header text changes.

18.5.6 Doxygen Guidelines

The DPDK API is documented using Doxygen comment annotations in the header files. Doxygen is a very powerful tool, it is extremely configurable and with a little effort can be used to create expressive documents. See the Doxygen website for full details on how to use it.

The following are some guidelines for use of Doxygen in the DPDK API documentation:

- New libraries that are documented with Doxygen should be added to the Doxygen configuration file: `doc/api/doxy-api.conf`. It is only required to add the directory that contains the files. It isn't necessary to explicitly name each file since the configuration matches all `rte_*.h` files in the directory.
- Use proper capitalization and punctuation in the Doxygen comments since they will become sentences in the documentation. This in particular applies to single line comments, which is the case the is most often forgotten.
- Use `@` style Doxygen commands instead of `\` style commands.
- Add a general description of each library at the head of the main header files:

```
/**
 * @file
 * RTE Mempool.
 *
 * A memory pool is an allocator of fixed-size object. It is
 * identified by its name, and uses a ring to store free objects.
 * ...
 */
```

- Document the purpose of a function, the parameters used and the return value:

```
/**
 * Try to take the lock.
 *
 * @param sl
 *   A pointer to the spinlock.
 * @return
 *   1 if the lock is successfully taken; 0 otherwise.
 */
int rte_spinlock_trylock(rte_spinlock_t *sl);
```

- Doxygen supports Markdown style syntax such as bold, italics, fixed width text and lists. For example the second line in the `devargs` parameter in the previous example will be rendered as:

  > The strings should be a pci address like `0000:01:00.0` or **virtual** device name like `net_pcap0`.

- Use `-` instead of `*` for lists within the Doxygen comment since the latter can get confused with the comment delimiter.
- Add an empty line between the function description, the `@params` and `@return` for readability.
- Place the `@params` description on separate line and indent it by 2 spaces. (It would be better to use no indentation since this is more common and also because checkpatch complains about leading whitespace in comments. However this is the convention used in the existing DPDK code.)
- Documented functions can be linked to simply by adding `()` to the function name:

```
/**
 * The functions exported by the application Ethernet API to setup
 * a device designated by its port identifier must be invoked in
 * the following order:
 *     - rte_eth_dev_configure()
 *     - rte_eth_tx_queue_setup()
 *     - rte_eth_rx_queue_setup()
 *     - rte_eth_dev_start()
 */
```

  In the API documentation the functions will be rendered as links, see the online section of the rte_ethdev.h docs that contains the above text.

- The `@see` keyword can be used to create a *see also* link to another file or library. This directive should be placed on one line at the bottom of the documentation section.

```
/**
 * ...
 *
 * Some text that references mempools.
 *
```

(continues on next page)

(continued from previous page)

```
 * @see eal_memzone.c
 */
```

- Doxygen supports two types of comments for documenting variables, constants and members: prefix and postfix:

```
/** This is a prefix comment. */
#define RTE_FOO_ERROR  0x023.

#define RTE_BAR_ERROR  0x024. /**< This is a postfix comment. */
```

- Postfix comments are preferred for struct members and constants if they can be documented in the same way:

```
struct rte_eth_stats {
    uint64_t ipackets; /**< Total number of received packets. */
    uint64_t opackets; /**< Total number of transmitted packets.*/
    uint64_t ibytes;   /**< Total number of received bytes. */
    uint64_t obytes;   /**< Total number of transmitted bytes. */
    uint64_t imissed;  /**< Total of RX missed packets. */
    uint64_t ibadcrc;  /**< Total of RX packets with CRC error. */
    uint64_t ibadlen;  /**< Total of RX packets with bad length. */
}
```

  Note: postfix comments should be aligned with spaces not tabs in accordance with the *DPDK Coding Style*.

- If a single comment type can't be used, due to line length limitations then prefix comments should be preferred. For example this section of the code contains prefix comments, postfix comments on the same line and postfix comments on a separate line:

```
/** Number of elements in the elt_pa array. */
uint32_t    pg_num __rte_cache_aligned;
uint32_t    pg_shift;     /**< LOG2 of the physical pages. */
uintptr_t   pg_mask;      /**< Physical page mask value. */
uintptr_t   elt_va_start;
/**< Virtual address of the first mempool object. */
uintptr_t   elt_va_end;
/**< Virtual address of the <size + 1> mempool object. */
phys_addr_t elt_pa[MEMPOOL_PG_NUM_DEFAULT];
/**< Array of physical page addresses for the mempool buffer. */
```

  This doesn't have an effect on the rendered documentation but it is confusing for the developer reading the code. It this case it would be clearer to use prefix comments throughout:

```
/** Number of elements in the elt_pa array. */
uint32_t    pg_num __rte_cache_aligned;
/** LOG2 of the physical pages. */
uint32_t    pg_shift;
/** Physical page mask value. */
uintptr_t   pg_mask;
/** Virtual address of the first mempool object. */
uintptr_t   elt_va_start;
/** Virtual address of the <size + 1> mempool object. */
uintptr_t   elt_va_end;
/** Array of physical page addresses for the mempool buffer. */
phys_addr_t elt_pa[MEMPOOL_PG_NUM_DEFAULT];
```

- Check for Doxygen warnings in new code by checking the API documentation build:

```
make doc-api-html >/dev/null
```

- Read the rendered section of the documentation that you have added for correctness, clarity and consistency with the surrounding text.

18.6 Contributing Code to DPDK

This document outlines the guidelines for submitting code to DPDK.

The DPDK development process is modeled (loosely) on the Linux Kernel development model so it is worth reading the Linux kernel guide on submitting patches: How to Get Your Change Into the Linux Kernel. The rationale for many of the DPDK guidelines is explained in greater detail in the kernel guidelines.

18.6.1 The DPDK Development Process

The DPDK development process has the following features:

- The code is hosted in a public git repository.
- There is a mailing list where developers submit patches.
- There are maintainers for hierarchical components.
- Patches are reviewed publicly on the mailing list.
- Successfully reviewed patches are merged to the repository.
- Patches should be sent to the target repository or sub-tree, see below.
- All sub-repositories are merged into main repository for `-rc1` and `-rc2` versions of the release.
- After the `-rc2` release all patches should target the main repository.

The mailing list for DPDK development is dev@dpdk.org. Contributors will need to register for the mailing list in order to submit patches. It is also worth registering for the DPDK Patchwork

If you are using the GitHub service, you can link your repository to the `travis-ci.org` build service. When you push patches to your GitHub repository, the travis service will automatically build your changes.

The development process requires some familiarity with the `git` version control system. Refer to the Pro Git Book for further information.

18.6.2 Source License

The DPDK uses the Open Source BSD-3-Clause license for the core libraries and drivers. The kernel components are GPL-2.0 licensed. DPDK uses single line reference to Unique License Identifiers in source files as defined by the Linux Foundation's SPDX project.

DPDK uses first line of the file to be SPDX tag. In case of *#!* scripts, SPDX tag can be placed in 2nd line of the file.

For example, to label a file as subject to the BSD-3-Clause license, the following text would be used:

```
SPDX-License-Identifier:  BSD-3-Clause
```

To label a file as dual-licensed with BSD-3-Clause and GPL-2.0 (e.g., for code that is shared between the kernel and userspace), the following text would be used:

```
SPDX-License-Identifier:  (BSD-3-Clause OR GPL-2.0)
```

Refer to `licenses/README` for more details.

18.6.3 Maintainers and Sub-trees

The DPDK maintenance hierarchy is divided into a main repository `dpdk` and sub-repositories `dpdk-next-*`.

There are maintainers for the trees and for components within the tree.

Trees and maintainers are listed in the `MAINTAINERS` file. For example:

```
Crypto Drivers
--------------
M: Some Name <some.name@email.com>
T: git://dpdk.org/next/dpdk-next-crypto

Intel AES-NI GCM PMD
M: Some One <some.one@email.com>
F: drivers/crypto/aesni_gcm/
F: doc/guides/cryptodevs/aesni_gcm.rst
```

Where:

- `M` is a tree or component maintainer.
- `T` is a repository tree.
- `F` is a maintained file or directory.

Additional details are given in the `MAINTAINERS` file.

The role of the component maintainers is to:

- Review patches for the component or delegate the review. The review should be done, ideally, within 1 week of submission to the mailing list.
- Add an `acked-by` to patches, or patchsets, that are ready for committing to a tree.
- Reply to questions asked about the component.

Component maintainers can be added or removed by submitting a patch to the `MAINTAINERS` file. Maintainers should have demonstrated a reasonable level of contributions or reviews to the component area. The maintainer should be confirmed by an `ack` from an established contributor. There can be more than one component maintainer if desired.

The role of the tree maintainers is to:

- Maintain the overall quality of their tree. This can entail additional review, compilation checks or other tests deemed necessary by the maintainer.
- Commit patches that have been reviewed by component maintainers and/or other contributors. The tree maintainer should determine if patches have been reviewed sufficiently.
- Ensure that patches are reviewed in a timely manner.
- Prepare the tree for integration.

- Ensure that there is a designated back-up maintainer and coordinate a handover for periods where the tree maintainer can't perform their role.

Tree maintainers can be added or removed by submitting a patch to the `MAINTAINERS` file. The proposer should justify the need for a new sub-tree and should have demonstrated a sufficient level of contributions in the area or to a similar area. The maintainer should be confirmed by an `ack` from an existing tree maintainer. Disagreements on trees or maintainers can be brought to the Technical Board.

The backup maintainer for the master tree should be selected from the existing sub-tree maintainers from the project. The backup maintainer for a sub-tree should be selected from among the component maintainers within that sub-tree.

18.6.4 Getting the Source Code

The source code can be cloned using either of the following:

main repository:

```
git clone git://dpdk.org/dpdk
git clone https://dpdk.org/git/dpdk
```

sub-repositories (list):

```
git clone git://dpdk.org/next/dpdk-next-*
git clone https://dpdk.org/git/next/dpdk-next-*
```

18.6.5 Make your Changes

Make your planned changes in the cloned `dpdk` repo. Here are some guidelines and requirements:

- Follow the *DPDK Coding Style* guidelines.
- If you add new files or directories you should add your name to the `MAINTAINERS` file.
- Initial submission of new PMDs should be prepared against a corresponding repo.
  - Thus, for example, initial submission of a new network PMD should be prepared against dpdk-next-net repo.
  - Likewise, initial submission of a new crypto or compression PMD should be prepared against dpdk-next-crypto repo.
  - For other PMDs and more info, refer to the `MAINTAINERS` file.
- New external functions should be added to the local `version.map` file. See the *ABI policy* and *ABI versioning* guides. New external functions should also be added in alphabetical order.
- Important changes will require an addition to the release notes in `doc/guides/rel_notes/`. See the *Release Notes section of the Documentation Guidelines* for details.
- Test the compilation works with different targets, compilers and options, see *Checking Compilation*.
- Don't break compilation between commits with forward dependencies in a patchset. Each commit should compile on its own to allow for `git bisect` and continuous integration testing.
- Add tests to the `app/test` unit test framework where possible.

- Add documentation, if relevant, in the form of Doxygen comments or a User Guide in RST format. See the *Documentation Guidelines*.

Once the changes have been made you should commit them to your local repo.

For small changes, that do not require specific explanations, it is better to keep things together in the same patch. Larger changes that require different explanations should be separated into logical patches in a patchset. A good way of thinking about whether a patch should be split is to consider whether the change could be applied without dependencies as a backport.

It is better to keep the related documentation changes in the same patch file as the code, rather than one big documentation patch at the end of a patchset. This makes it easier for future maintenance and development of the code.

As a guide to how patches should be structured run `git log` on similar files.

18.6.6 Commit Messages: Subject Line

The first, summary, line of the git commit message becomes the subject line of the patch email. Here are some guidelines for the summary line:

- The summary line must capture the area and the impact of the change.
- The summary line should be around 50 characters.
- The summary line should be lowercase apart from acronyms.
- It should be prefixed with the component name (use git log to check existing components). For example:

```
ixgbe: fix offload config option name

config: increase max queues per port
```

- Use the imperative of the verb (like instructions to the code base).
- Don't add a period/full stop to the subject line or you will end up two in the patch name: `dpdk_description..patch`.

The actual email subject line should be prefixed by `[PATCH]` and the version, if greater than v1, for example: `PATCH v2`. The is generally added by `git send-email` or `git format-patch`, see below.

If you are submitting an RFC draft of a feature you can use `[RFC]` instead of `[PATCH]`. An RFC patch doesn't have to be complete. It is intended as a way of getting early feedback.

18.6.7 Commit Messages: Body

Here are some guidelines for the body of a commit message:

- The body of the message should describe the issue being fixed or the feature being added. It is important to provide enough information to allow a reviewer to understand the purpose of the patch.
- When the change is obvious the body can be blank, apart from the signoff.
- The commit message must end with a `Signed-off-by:` line which is added using:

```
git commit --signoff # or -s
```

The purpose of the signoff is explained in the Developer's Certificate of Origin section of the Linux kernel guidelines.

Note: All developers must ensure that they have read and understood the Developer's Certificate of Origin section of the documentation prior to applying the signoff and submitting a patch.

- The signoff must be a real name and not an alias or nickname. More than one signoff is allowed.
- The text of the commit message should be wrapped at 72 characters.
- When fixing a regression, it is required to reference the id of the commit which introduced the bug, and put the original author of that commit on CC. You can generate the required lines using the following git alias, which prints the commit SHA and the author of the original code:

```
git config alias.fixline "log -1 --abbrev=12 --format='Fixes: %h (\"%s\")%nCc: %ae'"
```

The output of `git fixline <SHA>` must then be added to the commit message:

```
doc: fix some parameter description

Update the docs, fixing description of some parameter.

Fixes: abcdefgh1234 ("doc: add some parameter")
Cc: author@example.com

Signed-off-by: Alex Smith <alex.smith@example.com>
```

- When fixing an error or warning it is useful to add the error message and instructions on how to reproduce it.
- Use correct capitalization, punctuation and spelling.

In addition to the `Signed-off-by:` name the commit messages can also have tags for who reported, suggested, tested and reviewed the patch being posted. Please refer to the *Tested, Acked and Reviewed by* section.

Patch Fix Related Issues

Coverity is a tool for static code analysis. It is used as a cloud-based service used to scan the DPDK source code, and alert developers of any potential defects in the source code. When fixing an issue found by Coverity, the patch must contain a Coverity issue ID in the body of the commit message. For example:

```
doc: fix some parameter description

Update the docs, fixing description of some parameter.

Coverity issue: 12345
Fixes: abcdefgh1234 ("doc: add some parameter")
Cc: author@example.com

Signed-off-by: Alex Smith <alex.smith@example.com>
```

Bugzilla is a bug- or issue-tracking system. Bug-tracking systems allow individual or groups of developers effectively to keep track of outstanding problems with their product. When fixing an issue raised in Bugzilla, the patch must contain a Bugzilla issue ID in the body of the commit message. For example:

```
doc: fix some parameter description

Update the docs, fixing description of some parameter.

Bugzilla ID: 12345
Fixes: abcdefgh1234 ("doc: add some parameter")
Cc: author@example.com

Signed-off-by: Alex Smith <alex.smith@example.com>
```

Patch for Stable Releases

All fix patches to the master branch that are candidates for backporting should also be CCed to the stable@dpdk.org mailing list. In the commit message body the Cc: stable@dpdk.org should be inserted as follows:

```
doc: fix some parameter description

Update the docs, fixing description of some parameter.

Fixes: abcdefgh1234 ("doc: add some parameter")
Cc: stable@dpdk.org

Signed-off-by: Alex Smith <alex.smith@example.com>
```

For further information on stable contribution you can go to *Stable Contribution Guide*.

18.6.8 Creating Patches

It is possible to send patches directly from git but for new contributors it is recommended to generate the patches with `git format-patch` and then when everything looks okay, and the patches have been checked, to send them with `git send-email`.

Here are some examples of using `git format-patch` to generate patches:

```
# Generate a patch from the last commit.
git format-patch -1

# Generate a patch from the last 3 commits.
git format-patch -3

# Generate the patches in a directory.
git format-patch -3 -o ~/patch/

# Add a cover letter to explain a patchset.
git format-patch -3 -o ~/patch/ --cover-letter

# Add a prefix with a version number.
git format-patch -3 -o ~/patch/ -v 2
```

Cover letters are useful for explaining a patchset and help to generate a logical threading to the patches. Smaller notes can be put inline in the patch after the `---` separator, for example:

```
Subject: [PATCH] fm10k/base: add FM10420 device ids

Add the device ID for Boulder Rapids and Atwood Channel to enable
drivers to support those devices.

Signed-off-by: Alex Smith <alex.smith@example.com>
---

ADD NOTES HERE.

 drivers/net/fm10k/base/fm10k_api.c  | 6 ++++++
 drivers/net/fm10k/base/fm10k_type.h | 6 ++++++
 2 files changed, 12 insertions(+)
...
```

Version 2 and later of a patchset should also include a short log of the changes so the reviewer knows what has changed. This can be added to the cover letter or the annotations. For example:

```
---
v3:
* Fixed issued with version.map.

v2:
* Added i40e support.
* Renamed ethdev functions from rte_eth_ieee15888_*() to rte_eth_timesync_*()
  since 802.1AS can be supported through the same interfaces.
```

18.6.9 Checking the Patches

Patches should be checked for formatting and syntax issues using the `checkpatches.sh` script in the `devtools` directory of the DPDK repo. This uses the Linux kernel development tool `checkpatch.pl` which can be obtained by cloning, and periodically, updating the Linux kernel sources.

The path to the original Linux script must be set in the environment variable `DPDK_CHECKPATCH_PATH`.

Spell checking of commonly misspelled words can be enabled by downloading the codespell dictionary:

```
https://raw.githubusercontent.com/codespell-project/codespell/master/codespell_lib/data/
↪dictionary.txt
```

The path to the downloaded `dictionary.txt` must be set in the environment variable `DPDK_CHECKPATCH_CODESPELL`.

Environment variables required by the development tools, are loaded from the following files, in order of preference:

```
.develconfig
~/.config/dpdk/devel.config
/etc/dpdk/devel.config.
```

Once the environment variable is set, the script can be run as follows:

```
devtools/checkpatches.sh ~/patch/
```

The script usage is:

```
checkpatches.sh [-h] [-q] [-v] [patch1 [patch2] ...]]"
```

Where:

- `-h`: help, usage.
- `-q`: quiet. Don't output anything for files without issues.
- `-v`: verbose.
- `patchX`: path to one or more patches.

Then the git logs should be checked using the `check-git-log.sh` script.

The script usage is:

```
check-git-log.sh [range]
```

Where the range is a `git log` option.

18.6.10 Checking Compilation

Makefile System

Compilation of patches and changes should be tested using the `test-build.sh` script in the `devtools` directory of the DPDK repo:

```
devtools/test-build.sh x86_64-native-linux-gcc+next+shared
```

The script usage is:

```
test-build.sh [-h] [-jX] [-s] [config1 [config2] ...]]
```

Where:

- `-h`: help, usage.
- `-jX`: use X parallel jobs in "make".
- `-s`: short test with only first config and without examples/doc.
- `config`: default config name plus config switches delimited with a + sign.

Examples of configs are:

```
x86_64-native-linux-gcc
x86_64-native-linux-gcc+next+shared
x86_64-native-linux-clang+shared
```

The builds can be modified via the following environmental variables:

- `DPDK_BUILD_TEST_CONFIGS` (target1+option1+option2 target2)
- `DPDK_BUILD_TEST_DIR`
- `DPDK_DEP_CFLAGS`
- `DPDK_DEP_LDFLAGS`
- `DPDK_DEP_PCAP` (y/[n])

- DPDK_NOTIFY (notify-send)

These can be set from the command line or in the config files shown above in the *Checking the Patches*.

The recommended configurations and options to test compilation prior to submitting patches are:

```
x86_64-native-linux-gcc+shared+next
x86_64-native-linux-clang+shared
i686-native-linux-gcc

export DPDK_DEP_ZLIB=y
export DPDK_DEP_PCAP=y
export DPDK_DEP_SSL=y
```

Meson System

Compilation of patches is to be tested with `devtools/test-meson-builds.sh` script.

The script internally checks for dependencies, then builds for several combinations of compilation configuration. By default, each build will be put in a subfolder of the current working directory. However, if it is preferred to place the builds in a different location, the environment variable `DPDK_BUILD_TEST_DIR` can be set to that desired location. For example, setting `DPDK_BUILD_TEST_DIR=__builds` will put all builds in a single subfolder called "__builds" created in the current directory. Setting `DPDK_BUILD_TEST_DIR` to an absolute directory path e.g. `/tmp` is also supported.

18.6.11 Checking ABI compatibility

By default, ABI compatibility checks are disabled.

To enable them, a reference version must be selected via the environment variable `DPDK_ABI_REF_VERSION`.

The `devtools/test-build.sh` and `devtools/test-meson-builds.sh` scripts then build this reference version in a temporary directory and store the results in a subfolder of the current working directory. The environment variable `DPDK_ABI_REF_DIR` can be set so that the results go to a different location.

18.6.12 Sending Patches

Patches should be sent to the mailing list using `git send-email`. You can configure an external SMTP with something like the following:

```
[sendemail]
    smtpuser = name@domain.com
    smtpserver = smtp.domain.com
    smtpserverport = 465
    smtpencryption = ssl
```

See the Git send-email documentation for more details.

The patches should be sent to `dev@dpdk.org`. If the patches are a change to existing files then you should send them TO the maintainer(s) and CC `dev@dpdk.org`. The appropriate maintainer can be found in the `MAINTAINERS` file:

```
git send-email --to maintainer@some.org --cc dev@dpdk.org 000*.patch
```

Script `get-maintainer.sh` can be used to select maintainers automatically:

```
git send-email --to-cmd ./devtools/get-maintainer.sh --cc dev@dpdk.org 000*.patch
```

New additions can be sent without a maintainer:

```
git send-email --to dev@dpdk.org 000*.patch
```

You can test the emails by sending it to yourself or with the `--dry-run` option.

If the patch is in relation to a previous email thread you can add it to the same thread using the Message ID:

```
git send-email --to dev@dpdk.org --in-reply-to <1234-foo@bar.com> 000*.patch
```

The Message ID can be found in the raw text of emails or at the top of each Patchwork patch, for example. Shallow threading (`--thread --no-chain-reply-to`) is preferred for a patch series.

Once submitted your patches will appear on the mailing list and in Patchwork.

Experienced committers may send patches directly with `git send-email` without the `git format-patch` step. The options `--annotate` and `confirm = always` are recommended for checking patches before sending.

Backporting patches for Stable Releases

Sometimes a maintainer or contributor wishes, or can be asked, to send a patch for a stable release rather than mainline. In this case the patch(es) should be sent to `stable@dpdk.org`, not to `dev@dpdk.org`.

Given that there are multiple stable releases being maintained at the same time, please specify exactly which branch(es) the patch is for using `git send-email --subject-prefix='PATCH 16.11' ...` and also optionally in the cover letter or in the annotation.

18.6.13 The Review Process

Patches are reviewed by the community, relying on the experience and collaboration of the members to double-check each other's work. There are a number of ways to indicate that you have checked a patch on the mailing list.

Tested, Acked and Reviewed by

To indicate that you have interacted with a patch on the mailing list you should respond to the patch in an email with one of the following tags:

- Reviewed-by:
- Acked-by:
- Tested-by:
- Reported-by:
- Suggested-by:

The tag should be on a separate line as follows:

```
tag-here: Name Surname <email@address.com>
```

Each of these tags has a specific meaning. In general, the DPDK community follows the kernel usage of the tags. A short summary of the meanings of each tag is given here for reference:

`Reviewed-by:` is a strong statement that the patch is an appropriate state for merging without any remaining serious technical issues. Reviews from community members who are known to understand the subject area and to perform thorough reviews will increase the likelihood of the patch getting merged.

`Acked-by:` is a record that the person named was not directly involved in the preparation of the patch but wishes to signify and record their acceptance and approval of it.

`Tested-by:` indicates that the patch has been successfully tested (in some environment) by the person named.

`Reported-by:` is used to acknowledge person who found or reported the bug.

`Suggested-by:` indicates that the patch idea was suggested by the named person.

Steps to getting your patch merged

The more work you put into the previous steps the easier it will be to get a patch accepted. The general cycle for patch review and acceptance is:

1. Submit the patch.
2. Check the automatic test reports in the coming hours.
3. Wait for review comments. While you are waiting review some other patches.
4. Fix the review comments and submit a `v n+1` patchset:

   ```
   git format-patch -3 -v 2
   ```

5. Update Patchwork to mark your previous patches as "Superseded".
6. If the patch is deemed suitable for merging by the relevant maintainer(s) or other developers they will `ack` the patch with an email that includes something like:

   ```
   Acked-by: Alex Smith <alex.smith@example.com>
   ```

   Note: When acking patches please remove as much of the text of the patch email as possible. It is generally best to delete everything after the `Signed-off-by:` line.
7. Having the patch `Reviewed-by:` and/or `Tested-by:` will also help the patch to be accepted.
8. If the patch isn't deemed suitable based on being out of scope or conflicting with existing functionality it may receive a `nack`. In this case you will need to make a more convincing technical argument in favor of your patches.
9. In addition a patch will not be accepted if it doesn't address comments from a previous version with fixes or valid arguments.
10. It is the responsibility of a maintainer to ensure that patches are reviewed and to provide an `ack` or `nack` of those patches as appropriate.
11. Once a patch has been acked by the relevant maintainer, reviewers may still comment on it for a further two weeks. After that time, the patch should be merged into the relevant git tree for the next release. Additional notes and restrictions:

- Patches should be acked by a maintainer at least two days before the release merge deadline, in order to make that release.
- For patches acked with less than two weeks to go to the merge deadline, all additional comments should be made no later than two days before the merge deadline.
- After the appropriate time for additional feedback has passed, if the patch has not yet been merged to the relevant tree by the committer, it should be treated as though it had, in that any additional changes needed to it must be addressed by a follow-on patch, rather than rework of the original.
- Trivial patches may be merged sooner than described above at the tree committer's discretion.

18.7 DPDK Vulnerability Management Process

18.7.1 Scope

Only the main repositories (dpdk and dpdk-stable) of the core project are in the scope of this security process (including experimental APIs). If a stable branch is declared unmaintained (end of life), no fix will be applied.

All vulnerabilities are bugs, but not every bug is a vulnerability. Vulnerabilities compromise one or more of:

- Confidentiality (personal or corporate confidential data).
- Integrity (trustworthiness and correctness).
- Availability (uptime and service).

If in doubt, please consider the vulnerability as security sensitive. At worst, the response will be to report the bug through the usual channels.

18.7.2 Finding

There is no pro-active security engineering effort at the moment.

Please report any security issue you find in DPDK as described below.

18.7.3 Report

Do not use Bugzilla (unsecured). Instead, send GPG-encrypted emails to security@dpdk.org. Anyone can post to this list. In order to reduce the disclosure of a vulnerability in the early stages, membership of this list is intentionally limited to a small number of people.

It is additionally encouraged to GPG-sign one-on-one conversations as part of the security process.

As it is with any bug, the more information provided, the easier it will be to diagnose and fix. If you already have a fix, please include it with your report, as that can speed up the process considerably.

In the report, please note how you would like to be credited for discovering the issue and the details of any embargo you would like to impose.

If the vulnerability is not public yet, no patch or information should be disclosed publicly. If a fix is already published, the reporting process must be followed anyway, as described below.

18.7.4 Confirmation

Upon reception of the report, a security team member should reply to the reporter acknowledging that the report has been received.

The DPDK security team reviews the security vulnerability reported. Area experts not members of the security team may be involved in the process. In case the reported issue is not qualified as a security vulnerability, the security team will request the submitter to report it using the usual channel (Bugzilla). If qualified, the security team will assess which DPDK version are affected. A bugzilla ID (allocated in a reserved pool) is assigned to the vulnerability, and kept empty until public disclosure.

The security team calculates the severity score with CVSS calculator based on inputs from the reporter and its own assessment of the vulnerability, and agrees on the score with the reporter.

An embargo may be put in place depending on the severity of the vulnerability. If an embargo is decided, its duration should be suggested by the security team and negotiated with the reporter. Embargo duration between vulnerability confirmation and public disclosure should be between **one and ten weeks**. If an embargo is not required, the vulnerability may be fixed using the standard patch process, once a CVE number has been assigned.

The confirmation mail should be sent within **3 business days**.

Following information must be included in the mail:

- Confirmation
- CVSS severity and score
- Embargo duration
- Reporter credit
- Bug ID (empty and restricted for future reference)

18.7.5 CVE Request

The security team develops a security advisory document. The security team may, at its discretion, include the reporter (via "CC") in developing the security advisory document, but in any case should accept feedback from the reporter before finalizing the document. When the document is final, the security team needs to request a CVE identifier from a CNA.

The CVE request should be sent to secalert@redhat.com using GPG encrypted email (see contact details).

CVE Request Template with Embargo

```
A vulnerability was discovered in the DPDK project.
In order to ensure full traceability, we need a CVE number assigned
that we can attach to private and public notifications.
Please treat the following information as confidential during the embargo
until further public disclosure.

[PRODUCT]:
[VERSION]:
[PROBLEMTYPE]:
[SEVERITY]:
[REFERENCES]: { bug_url }
[DESCRIPTION]:
```

(continues on next page)

(continued from previous page)

```
Thanks
{ DPDK_security_team_member }, on behalf of the DPDK security team
```

CVE Request Template without Embargo

```
A vulnerability was discovered in the DPDK project.
In order to ensure full traceability, we need a CVE number assigned
that we can attach to private and public notifications.

[PRODUCT]:
[VERSION]:
[PROBLEMTYPE]:
[SEVERITY]:
[REFERENCES]: { bug_url }
[DESCRIPTION]:

Thanks
{ DPDK_security_team_member }, on behalf of the DPDK security team
```

18.7.6 Fix Development and Review

If the fix is already published, this step is skipped, and the pre-release disclosure is replaced with the private disclosure, as described below. It must not be considered as the standard process.

This step may be started in parallel with CVE creation. The patches fixing the vulnerability are developed and reviewed by the security team and by elected area experts that agree to maintain confidentiality.

The CVE id and the bug id must be referenced in the patch.

Backports to the identified affected versions are done once the fix is ready.

18.7.7 Pre-Release Disclosure

When the fix is ready, the security advisory and patches are sent to downstream stakeholders (security-prerelease@dpdk.org), specifying the date and time of the end of the embargo. The communicated public disclosure date should be **less than one week**

Downstream stakeholders are expected not to deploy or disclose patches until the embargo is passed, otherwise they will be removed from the list.

Downstream stakeholders (in security-prerelease list), are:

- Operating system vendors known to package DPDK
- Major DPDK users, considered trustworthy by the technical board, who have made the request to techboard@dpdk.org

The *OSS security private mailing list mailto:distros@vs.openwall.org>* will also be contacted one week before the end of the embargo, as indicated by *the OSS-security process <https://oss-security.openwall.org/wiki/mailing-lists/distros>* and using the PGP key listed on the same page, describing the details of the vulnerability and sharing the patch[es]. Distributions and major vendors follow this private mailing list, and it functions as a single point of contact for embargoed advance notices for open source projects.

The security advisory will be based on below template, and will be sent signed with a security team's member GPG key.

Pre-Release Mail Template

```
This is an advance warning of a vulnerability discovered in DPDK,
to give you, as downstream stakeholders, a chance to coordinate
the release of fixes and reduce the vulnerability window.
Please treat the following information as confidential until
the proposed public disclosure date.

{ impact_description }

Proposed patches are attached.
Unless a flaw is discovered in them, these patches will be merged
to { branches } on the public disclosure date.

CVE: { cve_id }
Severity: { severity }
CVSS scores: { cvss_scores }

Proposed public disclosure date/time: { disclosure_date } at 15:00 UTC.
Please do not make the issue public (or release public patches)
before this coordinated embargo date.
```

If the issue is leaked during the embargo, the same procedure is followed with only a few days delay between the pre-release and the public disclosure.

18.7.8 Private Disclosure

If a vulnerability is unintentionally already fixed in the public repository, a security advisory is sent to downstream stakeholders (security-prerelease@dpdk.org), giving few days to prepare for updating before the public disclosure.

Private Disclosure Mail Template

```
This is a warning of a vulnerability discovered in DPDK,
to give you, as downstream stakeholders, a chance to coordinate
the deployment of fixes before a CVE is public.

Please treat the following information as confidential until
the proposed public disclosure date.

{ impact_description }

Commits: { commit_ids with branch number }

CVE: { cve_id }
Severity: { severity }
CVSS scores: { cvss_scores }

Proposed public disclosure date/time: { disclosure_date }.
Please do not make the vulnerability information public
before this coordinated embargo date.
```

18.7.9 Public Disclosure

On embargo expiration, following tasks will be done simultaneously:

- The assigned bug is filled by a member of the security team, with all relevant information, and it is made public.
- The patches are pushed to the appropriate branches.
- For long and short term stable branches fixed, new versions should be released.

Releases on Monday to Wednesday are preferred, so that system administrators do not have to deal with security updates over the weekend.

The security advisory is posted to announce@dpdk.org and to *the public OSS-security mailing list <mailto:oss-security@lists.openwall.com>* as soon as the patches are pushed to the appropriate branches.

Patches are then sent to dev@dpdk.org and stable@dpdk.org accordingly.

Release Mail Template

```
A vulnerability was fixed in DPDK.
Some downstream stakeholders were warned in advance
in order to coordinate the release of fixes
and reduce the vulnerability window.

{ impact_description }

Commits: { commit_ids with branch number }

CVE: { cve_id }
Bugzilla: { bug_url }
Severity: { severity }
CVSS scores: { cvss_scores }
```

18.7.10 References

- A minimal security response process
- fd.io Vulnerability Management
- Open Daylight Vulnerability Management
- CVE Assignment Information Format

18.8 DPDK Stable Releases and Long Term Support

This section sets out the guidelines for the DPDK Stable Releases and the DPDK Long Term Support releases (LTS).

18.8.1 Introduction

The purpose of the DPDK Stable Releases is to maintain releases of DPDK with backported fixes over an extended period of time. This provides downstream consumers of DPDK with a stable target on which to base applications or packages.

The Long Term Support release (LTS) is a designation applied to a Stable Release to indicate longer term support.

18.8.2 Stable Releases

Any release of DPDK can be designated as a Stable Release if a maintainer volunteers to maintain it and there is a commitment from major contributors to validate it before releases. If a release is to be designated as a Stable Release, it should be done by 1 month after the master release.

A Stable Release is used to backport fixes from an `N` release back to an `N-1` release, for example, from 16.11 to 16.07.

The duration of a stable is one complete release cycle (3 months). It can be longer, up to 1 year, if a maintainer continues to support the stable branch, or if users supply backported fixes, however the explicit commitment should be for one release cycle.

The release cadence is determined by the maintainer based on the number of bugfixes and the criticality of the bugs. Releases should be coordinated with the validation engineers to ensure that a tagged release has been tested.

18.8.3 LTS Release

A stable release can be designated as an LTS release based on community agreement and a commitment from a maintainer. The current policy is that each year's November (X.11) release will be maintained as an LTS for 2 years.

After the X.11 release, an LTS branch will be created for it at https://git.dpdk.org/dpdk-stable where bugfixes will be backported to.

A LTS release may align with the declaration of a new major ABI version, please read the *ABI Policy* for more information.

It is anticipated that there will be at least 4 releases per year of the LTS or approximately 1 every 3 months. However, the cadence can be shorter or longer depending on the number and criticality of the backported fixes. Releases should be coordinated with the validation engineers to ensure that a tagged release has been tested.

For a list of the currently maintained stable/LTS branches please see the latest stable roadmap.

At the end of the 2 years, a final X.11.N release will be made and at that point the LTS branch will no longer be maintained with no further releases.

18.8.4 What changes should be backported

Backporting should be limited to bug fixes. All patches accepted on the master branch with a Fixes: tag should be backported to the relevant stable/LTS branches, unless the submitter indicates otherwise. If there are exceptions, they will be discussed on the mailing lists.

Fixes suitable for backport should have a `Cc: stable@dpdk.org` tag in the commit message body as follows:

```
doc: fix some parameter description

Update the docs, fixing description of some parameter.

Fixes: abcdefgh1234 ("doc: add some parameter")
Cc: stable@dpdk.org

Signed-off-by: Alex Smith <alex.smith@example.com>
```

Fixes not suitable for backport should not include the `Cc: stable@dpdk.org` tag.

Features should not be backported to stable releases. It may be acceptable, in limited cases, to back port features for the LTS release where:

- There is a justifiable use case (for example a new PMD).
- The change is non-invasive.
- The work of preparing the backport is done by the proposer.
- There is support within the community.

18.8.5 The Stable Mailing List

The Stable and LTS release are coordinated on the stable@dpdk.org mailing list.

All fix patches to the master branch that are candidates for backporting should also be CCed to the stable@dpdk.org mailing list.

18.8.6 Releasing

A Stable Release will be released by:

- Tagging the release with YY.MM.n (year, month, number).
- Uploading a tarball of the release to dpdk.org.
- Sending an announcement to the announce@dpdk.org list.

Stable releases are available on the dpdk.org download page.

18.9 Patch Cheatsheet

Fig. 18.3: Cheat sheet for submitting patches to dev@dpdk.org

CHAPTER

NINETEEN

RELEASE NOTES

19.1 DPDK Release 20.05

19.1.1 New Features

- **Added Trace Library and Tracepoints.**

  Added a native implementation of the "common trace format" (CTF) based trace library. This allows the user add tracepoints in an application/library to get runtime trace/debug information for control, and fast APIs with minimum impact on fast path performance. Typical trace overhead is ~20 cycles and instrumentation overhead is 1 cycle. Added tracepoints in `EAL`, `ethdev`, `cryptodev`, `eventdev` and `mempool` libraries for important functions.

- **Added APIs for RCU defer queues.**

  Added APIs to create and delete defer queues. Additional APIs are provided to enqueue a deleted resource and reclaim the resource in the future. These APIs help an application use lock-free data structures with less effort.

- **Added new API for rte_ring.**

  - Introduced new synchronization modes for `rte_ring`.

    Introduced new optional MT synchronization modes for `rte_ring`: Relaxed Tail Sync (RTS) mode and Head/Tail Sync (HTS) mode. With these modes selected, `rte_ring` shows significant improvements for average enqueue/dequeue times on overcommitted systems.

  - Added peek style API for `rte_ring`.

    For rings with producer/consumer in `RTE_RING_SYNC_ST`, `RTE_RING_SYNC_MT_HTS` mode, provide the ability to split enqueue/dequeue operation into two phases (enqueue/dequeue start and enqueue/dequeue finish). This allows the user to inspect objects in the ring without removing them (aka MT safe peek).

- **Added flow aging support.**

  Added flow aging support to detect and report aged-out flows, including:

  - Added new action: `RTE_FLOW_ACTION_TYPE_AGE` to set the timeout and the application flow context for each flow.
  - Added new event: `RTE_ETH_EVENT_FLOW_AGED` for the driver to report that there are new aged-out flows.
  - Added new query: `rte_flow_get_aged_flows` to get the aged-out flows contexts from the port.

- **ethdev: Added a new value to link speed for 200Gbps.**

  Added a new ethdev value to for link speeds of 200Gbps.

- **Updated the Amazon ena driver.**

  Updated the ena PMD with new features and improvements, including:

  - Added support for large LLQ (Low-latency queue) headers.
  - Added Tx drops as a new extended driver statistic.
  - Added support for accelerated LLQ mode.
  - Handling of the 0 length descriptors on the Rx path.

- **Updated Broadcom bnxt driver.**

  Updated the Broadcom bnxt driver with new features and improvements, including:

  - Added support for host based flow table management.
  - Added flow counters to extended stats.
  - Added PCI function stats to extended stats.

- **Updated Hisilicon hns3 driver.**

  Updated Hisilicon hns3 driver with new features and improvements, including:

  - Added support for TSO.
  - Added support for configuring promiscuous and allmulticast mode for VF.

- **Added a new driver for Intel Foxville I225 devices.**

  Added the new `igc` net driver for Intel Foxville I225 devices. See the *IGC Poll Mode Driver* NIC guide for more details on this new driver.

- **Updated Intel i40e driver.**

  Updated i40e PMD with new features and improvements, including:

  - Enabled MAC address as FDIR input set for ipv4-other, ipv4-udp and ipv4-tcp.
  - Added support for RSS using L3/L4 source/destination only.
  - Added support for setting hash function in rte flow.

- **Updated the Intel iavf driver.**

  Update the Intel iavf driver with new features and improvements, including:

  - Added generic filter support.
  - Added advanced iavf with FDIR capability.
  - Added advanced RSS configuration for VFs.

- **Updated the Intel ice driver.**

  Updated the Intel ice driver with new features and improvements, including:

  - Added support for DCF (Device Config Function) feature.
  - Added switch filter support for Intel DCF.

- **Updated Marvell OCTEON TX2 ethdev driver.**

  Updated Marvell OCTEON TX2 ethdev driver with traffic manager support, including:

  - Hierarchical Scheduling with DWRR and SP.
  - Single rate - Two color, Two rate - Three color shaping.

- **Updated Mellanox mlx5 driver.**

  Updated Mellanox mlx5 driver with new features and improvements, including:

  - Added support for matching on IPv4 Time To Live and IPv6 Hop Limit.
  - Added support for creating Relaxed Ordering Memory Regions.
  - Added support for configuring Hairpin queue data buffer size.
  - Added support for jumbo frame size (9K MTU) in Multi-Packet RQ mode.
  - Removed flow rules caching for memory saving and compliance with ethdev API.
  - Optimized the memory consumption of flows.
  - Added support for flow aging based on hardware counters.
  - Added support for flow patterns with wildcard VLAN items (without VID value).
  - Updated support for matching on GTP headers, added match on GTP flags.

- **Added Chacha20-Poly1305 algorithm to Cryptodev API.**

  Added support for Chacha20-Poly1305 AEAD algorithm in Cryptodev.

- **Updated the AESNI MB crypto PMD.**

  - Added support for intel-ipsec-mb version 0.54.
  - Updated the AESNI MB PMD with AES-256 DOCSIS algorithm.
  - Added support for synchronous Crypto burst API.

- **Updated the AESNI GCM crypto PMD.**

  Added support for intel-ipsec-mb version 0.54.

- **Updated the ZUC crypto PMD.**

  - Added support for intel-ipsec-mb version 0.54.
  - Updated the PMD to support Multi-buffer ZUC-EIA3, improving performance significantly, when using intel-ipsec-mb version 0.54

- **Updated the SNOW3G crypto PMD.**

  Added support for intel-ipsec-mb version 0.54.

- **Updated the KASUMI crypto PMD.**

  Added support for intel-ipsec-mb version 0.54.

- **Updated the QuickAssist Technology (QAT) Crypto PMD.**

  - Added handling of mixed crypto algorithms in QAT PMD for GEN2.

Enabled handling of mixed algorithms in encrypted digest hash-cipher (generation) and cipher-hash (verification) requests in QAT PMD when running on GEN2 QAT hardware with particular firmware versions (GEN3 support was added in DPDK 20.02).

- Added plain SHA-1, 224, 256, 384, 512 support to QAT PMD.

  Added support for plain SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 hashes to QAT PMD.

- Added AES-GCM/GMAC J0 support to QAT PMD.

  Added support for AES-GCM/GMAC J0 to Intel QuickAssist Technology PMD. The user can use this feature by passing a zero length IV in the appropriate xform. For more information refer to the doxygen comments in `rte_crypto_sym.h` for `J0`.

- Updated the QAT PMD for AES-256 DOCSIS.

  Added AES-256 DOCSIS algorithm support to the QAT PMD.

- **Updated the QuickAssist Technology (QAT) Compression PMD.**

  Added special buffer handling when the internal QAT intermediate buffer is too small for the Huffman dynamic compression operation. Instead of falling back to fixed compression, the operation is now split into multiple smaller dynamic compression requests (which are possible to execute on QAT) and their results are then combined and copied into the output buffer. This is not possible if any checksum calculation was requested - in such cases the code falls back to fixed compression as before.

- **Updated the turbo_sw bbdev PMD.**

  Added support for large size code blocks which do not fit in one mbuf segment.

- **Added Intel FPGA_5GNR_FEC bbdev PMD.**

  Added a new `fpga_5gnr_fec` bbdev driver for the Intel® FPGA PAC (Programmable Acceleration Card) N3000. See the *Intel(R) FPGA 5GNR FEC Poll Mode Driver* BBDEV guide for more details on this new driver.

- **Updated the DSW event device.**

  Updated the DSW PMD with new features and improvements, including:

  - Improved flow migration mechanism, allowing faster and more accurate load balancing.
  - Improved behavior on high-core count systems.
  - Reduced latency in low-load situations.
  - Extended DSW xstats with migration and load-related statistics.

- **Updated ipsec-secgw sample application.**

  Updated the `ipsec-secgw` sample application with the following features:

  - Updated the application to add event based packet processing. The worker thread(s) would receive events and submit them back to the event device after the processing. This way, multicore scaling and HW assisted scheduling is achieved by making use of the event device capabilities. The event mode currently only supports inline IPsec protocol offload.
  - Updated the application to support key sizes for AES-192-CBC, AES-192-GCM, AES-256-GCM algorithms.

  - Added IPsec inbound load-distribution support for the application using NIC load distribution feature (Flow Director).

- **Updated Telemetry Library.**

  The updated Telemetry library has been significantly improved in relation to the original version to make it more accessible and scalable:

  - It now enables DPDK libraries and applications to provide their own specific telemetry information, rather than being limited to what could be reported through the metrics library.
  - It is no longer dependent on the external Jansson library, which allows Telemetry be enabled by default.
  - The socket handling has been simplified making it easier for clients to connect and retrieve information.

- **Added the rte_graph library.**

  The Graph architecture abstracts the data processing functions as `nodes` and `links` them together to create a complex `graph` to enable reusable/modular data processing functions. The graph library provides APIs to enable graph framework operations such as create, lookup, dump and destroy on graph and node operations such as clone, edge update, and edge shrink, etc. The API also allows the creation of a stats cluster to monitor per graph and per node statistics.

- **Added the rte_node library.**

  Added the `rte_node` library that consists of nodes used by the `rte_graph` library. Each node performs a specific packet processing function based on the application configuration.

  The following nodes are added:

  - Null node: A skeleton node that defines the general structure of a node.
  - Ethernet device node: Consists of Ethernet Rx/Tx nodes as well as Ethernet control APIs.
  - IPv4 lookup node: Consists of IPv4 extract and LPM lookup node. Routes can be configured by the application through the `rte_node_ip4_route_add` function.
  - IPv4 rewrite node: Consists of IPv4 and Ethernet header rewrite functionality that can be configured through the `rte_node_ip4_rewrite_add` function.
  - Packet drop node: Frees the packets received to their respective mempool.

- **Added new l3fwd-graph sample application.**

  Added an example application `l3fwd-graph`. This demonstrates the usage of the graph library and node library for packet processing. In addition to the library usage demonstration, this application can be used for performance comparison of the existing `l3fwd` (static code without any nodes) with the modular `l3fwd-graph` approach.

- **Updated the testpmd application.**

  Added a new cmdline option `--rx-mq-mode` which can be used to test PMD's behaviour on handling Rx mq mode.

- **Added support for GCC 10.**

  Added support for building with GCC 10.1.

19.1.2 API Changes

- mempool: The API of `rte_mempool_populate_iova()` and `rte_mempool_populate_virt()` changed to return 0 instead of `-EINVAL` when there is not enough room to store one object.

19.1.3 ABI Changes

- No ABI change that would break compatibility with DPDK 20.02 and 19.11.

19.1.4 Tested Platforms

- Intel® platforms with Broadcom® NICs combinations
  - CPU:
    - Intel® Xeon® Gold 6154 CPU @ 3.00GHz
    - Intel® Xeon® CPU E5-2650 v2 @ 2.60GHz
    - Intel® Xeon® CPU E5-2667 v3 @ 3.20GHz
    - Intel® Xeon® Gold 6142 CPU @ 2.60GHz
    - Intel® Xeon® Silver 4110 CPU @ 2.10GHz
  - OS:
    - Red Hat Enterprise Linux Server release 8.1
    - Red Hat Enterprise Linux Server release 7.6
    - Red Hat Enterprise Linux Server release 7.5
    - Ubuntu 16.04
    - Centos 8.1
    - Centos 7.7
  - upstream kernel:
    - Linux 5.3
  - NICs:
    - Broadcom® NetXtreme-E® Series P225p (2x25G)
      - Host interface: PCI Express 3.0 x8
      - Firmware version: 214.4.81.0 and above
    - Broadcom® NetXtreme-E® Series P425p (4x25G)
      - Host interface: PCI Express 3.0 x16
      - Firmware version: 216.4.259.0 and above
    - Broadcom® NetXtreme-E® Series P2100G (2x100G)
      - Host interface: PCI Express 3.0 x16

            - Firmware version: 216.1.259.0 and above
        - Broadcom® NetXtreme-E® Series P425p (4x25G)
            - Host interface: PCI Express 4.0 x16
            - Firmware version: 216.1.259.0 and above
        - Broadcom® NetXtreme-E® Series P2100G (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Firmware version: 216.1.259.0 and above
- Intel® platforms with Intel® NICs combinations
    - CPU
        - Intel® Atom™ CPU C3758 @ 2.20GHz
        - Intel® Atom™ CPU C3858 @ 2.00GHz
        - Intel® Atom™ CPU C3958 @ 2.00GHz
        - Intel® Xeon® CPU D-1541 @ 2.10GHz
        - Intel® Xeon® CPU D-1553N @ 2.30GHz
        - Intel® Xeon® CPU E5-2680 0 @ 2.70GHz
        - Intel® Xeon® CPU E5-2680 v2 @ 2.80GHz
        - Intel® Xeon® CPU E5-2699 v3 @ 2.30GHz
        - Intel® Xeon® CPU E5-2699 v4 @ 2.20GHz
        - Intel® Xeon® Gold 5218N CPU @ 2.30GHz
        - Intel® Xeon® Gold 6139 CPU @ 2.30GHz
        - Intel® Xeon® Gold 6252N CPU @ 2.30GHz
        - Intel® Xeon® Platinum 8180 CPU @ 2.50GHz
        - Intel® Xeon® Platinum 8280M CPU @ 2.70GHz
    - OS:
        - CentOS 7.7
        - CentOS 8.0
        - Fedora 32
        - FreeBSD 12.1
        - OpenWRT 19.07
        - Red Hat Enterprise Linux Server release 8.0
        - Red Hat Enterprise Linux Server release 7.7
        - Suse15 SP1
        - Ubuntu 16.04
        - Ubuntu 18.04

    * Ubuntu 20.04

  - NICs:

    * Intel® 82599ES 10 Gigabit Ethernet Controller
      - Firmware version: 0x61bf0001
      - Device id (pf/vf): 8086:10fb / 8086:10ed
      - Driver version: 5.6.5 (ixgbe)
    * Intel® Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      - Firmware version: 0x800003e7
      - Device id (pf/vf): 8086:15ad / 8086:15a8
      - Driver version: 5.1.0-k (ixgbe)
    * Intel® Corporation Ethernet Controller 10G X550T
      - Firmware version: 0x80000482
      - Device id (pf): 8086:1563
      - Driver version: 5.6.5 (ixgbe)
    * Intel® Ethernet Converged Network Adapter X710-DA4 (4x10G)
      - Firmware version: 7.20 0x800079e8 1.2585.0
      - Device id (pf/vf): 8086:1572 / 8086:154c
      - Driver version: 2.11.29 (i40e)
    * Intel® Corporation Ethernet Connection X722 for 10GbE SFP+ (4x10G)
      - Firmware version: 4.11 0x80001def 1.1999.0
      - Device id (pf/vf): 8086:37d0 / 8086:37cd
      - Driver version: 2.11.29 (i40e)
    * Intel® Corporation Ethernet Connection X722 for 10GBASE-T (2x10G)
      - Firmware version: 4.10 0x80001a7a
      - Device id (pf/vf): 8086:37d2 / 8086:37cd
      - Driver version: 2.11.29 (i40e)
    * Intel® Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
      - Firmware version: 7.30 0x800080a2 1.2658.0
      - Device id (pf/vf): 8086:158b / 8086:154c
      - Driver version: 2.11.27_rc13 (i40e)
    * Intel® Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
      - Firmware version: 7.30 0x800080ab 1.2658.0
      - Device id (pf/vf): 8086:1583 / 8086:154c
      - Driver version: 2.11.27_rc13 (i40e)

        * Intel® Corporation I350 Gigabit Network Connection
            - Firmware version: 1.63, 0x80000cbc
            - Device id (pf/vf): 8086:1521 / 8086:1520
            - Driver version: 5.4.0-k (igb)
        * Intel® Corporation I210 Gigabit Network Connection
            - Firmware version: 3.25, 0x800006eb
            - Device id (pf): 8086:1533
            - Driver version: 5.6.5(igb)
        * Intel® Ethernet Controller 10-Gigabit X540-AT2
            - Firmware version: 0x800005f9
            - Device id (pf): 8086:1528
            - Driver version: 5.1.0-k(ixgbe)
        * Intel® Ethernet Converged Network Adapter X710-T2L
            - Firmware version: 7.30 0x80008061 1.2585.0
            - Device id (pf): 8086:15ff
            - Driver version: 2.11.27_rc13(i40e)
- Intel® platforms with Mellanox® NICs combinations
    - CPU:
        * Intel® Xeon® Gold 6154 CPU @ 3.00GHz
        * Intel® Xeon® CPU E5-2697A v4 @ 2.60GHz
        * Intel® Xeon® CPU E5-2697 v3 @ 2.60GHz
        * Intel® Xeon® CPU E5-2680 v2 @ 2.80GHz
        * Intel® Xeon® CPU E5-2650 v4 @ 2.20GHz
        * Intel® Xeon® CPU E5-2640 @ 2.50GHz
        * Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz
    - OS:
        * Red Hat Enterprise Linux Server release 7.5 (Maipo)
        * Red Hat Enterprise Linux Server release 7.4 (Maipo)
        * Red Hat Enterprise Linux Server release 7.3 (Maipo)
        * Red Hat Enterprise Linux Server release 7.2 (Maipo)
        * Ubuntu 18.04
        * Ubuntu 16.04
    - OFED:
        * MLNX_OFED 4.7-3.2.9.0

    - MLNX_OFED 5.0-2.1.8.0 and above
  - upstream kernel:
    - Linux 5.7.0-rc5 and above
  - rdma-core:
    - rdma-core-29.0-1 and above
  - NICs:
    - Mellanox® ConnectX®-3 Pro 40G MCX354A-FCC_Ax (2x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1007
      - Firmware version: 2.42.5000
    - Mellanox® ConnectX®-3 Pro 40G MCX354A-FCCT (2x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1007
      - Firmware version: 2.42.5000
    - Mellanox® ConnectX®-4 Lx 25G MCX4121A-ACAT (2x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1015
      - Firmware version: 14.27.2008 and above
    - Mellanox® ConnectX®-4 Lx 50G MCX4131A-GCAT (1x50G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1015
      - Firmware version: 14.27.2008 and above
    - Mellanox® ConnectX®-5 100G MCX516A-CCAT (2x100G)
      - Host interface: PCI Express 3.0 x16
      - Device ID: 15b3:1017
      - Firmware version: 16.27.2008 and above
    - Mellanox® ConnectX®-5 100G MCX556A-ECAT (2x100G)
      - Host interface: PCI Express 3.0 x16
      - Device ID: 15b3:1017
      - Firmware version: 16.27.2008 and above
    - Mellanox® ConnectX®-5 100G MCX556A-EDAT (2x100G)
      - Host interface: PCI Express 3.0 x16
      - Device ID: 15b3:1017
      - Firmware version: 16.27.2008 and above

        * Mellanox® ConnectX®-5 Ex EN 100G MCX516A-CDAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1019
            - Firmware version: 16.27.2008 and above
        * Mellanox® ConnectX®-6 Dx EN 100G MCX623106AN-CDAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:101d
            - Firmware version: 22.27.2008 and above
- IBM Power 9 platforms with Mellanox® NICs combinations
    - CPU:
        * POWER9 2.2 (pvr 004e 1202) 2300MHz
    - OS:
        * Red Hat Enterprise Linux Server release 7.6
    - NICs:
        * Mellanox® ConnectX®-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.27.2008
        * Mellanox® ConnectX®-6 Dx 100G MCX623106AN-CDAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:101d
            - Firmware version: 22.27.2008
    - OFED:
        * MLNX_OFED 5.0-2.1.8.0
- ARMv8 SoC combinations from Marvell (with integrated NICs)
    - SoC:
        * CN83xx, CN96xx, CN93xx
    - OS (Based on Marvell OCTEON TX SDK-10.3.2.0-PR12):
        * Arch Linux
        * Buildroot 2018.11
        * Ubuntu 16.04.1 LTS
        * Ubuntu 16.10
        * Ubuntu 18.04.1
        * Ubuntu 19.04

19.2 DPDK Release 20.02

19.2.1 New Features

- **Added Wait Until Equal API.**

  A new API has been added to wait for a memory location to be updated with a 16-bit, 32-bit, 64-bit value.

- **Added rte_ring_xxx_elem APIs.**

  New APIs have been added to support rings with custom element size.

- **Added mbuf pool with pinned external memory.**

  Added support of mbuf with data buffer allocated in an external device memory.

- **Updated rte_flow api to support L2TPv3 over IP flows.**

  Added support for new flow item to handle L2TPv3 over IP rte_flow patterns.

- **Added DSCP rewrite action.**

  New actions `RTE_FLOW_ACTION_TYPE_SET_IPV[4/6]_DSCP` have been added to support rewrite the DSCP field in the IP header.

- **Added IONIC net PMD.**

  Added the new `ionic` net driver for Pensando Ethernet Network Adapters. See the *IONIC Driver* NIC guide for more details on this new driver.

- **Updated Broadcom bnxt driver.**

  Updated Broadcom bnxt driver with new features and improvements, including:

  - Added support for MARK action.

- **Updated Hisilicon hns3 driver.**

  Updated Hisilicon hns3 driver with new features and improvements, including:

  - Added support for Rx interrupt.
  - Added support setting VF MAC address by PF driver.

- **Updated the Intel ice driver.**

  Updated the Intel ice driver with new features and improvements, including:

  - Added support for MAC rules on a specific port.
  - Added support for MAC/VLAN with TCP/UDP in switch rule.
  - Added support for 1/10G device.
  - Added support for API `rte_eth_tx_done_cleanup`.

- **Updated Intel iavf driver.**

  Updated iavf PMD with new features and improvements, including:

  - Added more supported device IDs.
  - Updated virtual channel to latest AVF spec.

- **Updated the Intel ixgbe driver.**

  Updated ixgbe PMD with new features and improvements, including:

  - Added support for API `rte_eth_tx_done_cleanup()`.
  - Added support setting VF MAC address by PF driver.
  - Added support for setting the link to specific speed.

- **Updated Intel i40e driver.**

  Updated i40e PMD with new features and improvements, including:

  - Added support for L2TPv3 over IP profiles which can be programmed by the dynamic device personalization (DDP) process.
  - Added support for ESP-AH profiles which can be programmed by the dynamic device personalization (DDP) process.
  - Added PF support Malicious Device Drive event catch and notify.
  - Added LLDP support.
  - Extended PHY access AQ cmd.
  - Added support for reading LPI counters.
  - Added support for Energy Efficient Ethernet.
  - Added support for API `rte_eth_tx_done_cleanup()`.
  - Added support for VF multiple queues interrupt.
  - Added support for setting the link to specific speed.

- **Updated Mellanox mlx5 driver.**

  Updated Mellanox mlx5 driver with new features and improvements, including:

  - Added support for the mbufs with external pinned buffers.
  - Added support for RSS using L3/L4 source/destination only.
  - Added support for matching on GTP tunnel header item.
  - Removed limitation of matching on tagged/untagged packets (when using DV flow engine).
  - Added support for IPv4/IPv6 DSCP rewrite action.
  - Added BlueField-2 integrated ConnectX-6 Dx device support.

- **Add new vDPA PMD based on Mellanox devices.**

  Added a new Mellanox vDPA (`mlx5_vdpa`) PMD. See the *MLX5 vDPA driver* guide for more details on this driver.

- **Added support for virtio-PMD notification data.**

  Added support for virtio-PMD notification data so that the driver passes extra data (besides identifying the virtqueue) in its device notifications, expanding the notifications to include the avail index and avail wrap counter (When split ring is used, the avail wrap counter is not included in the notification data).

- **Updated testpmd application.**

  Added support for ESP and L2TPv3 over IP rte_flow patterns to the testpmd application.

- **Added algorithms to cryptodev API.**

  Added new algorithms to the cryptodev API:

  - ECDSA (Elliptic Curve Digital Signature Algorithm) is added to asymmetric crypto library specifications.
  - ECPM (Elliptic Curve Point Multiplication) is added to asymmetric crypto library specifications.

- **Added synchronous Crypto burst API.**

  A new API has been introduced in the crypto library to handle synchronous cryptographic operations allowing it to achieve performance gains for cryptodevs which use CPU based acceleration, such as Intel AES-NI. An implementation for aesni_gcm cryptodev is provided. The IPsec example application and ipsec library itself were changed to allow utilization of this new feature.

- **Added handling of mixed algorithms in encrypted digest requests in QAT PMD.**

  Added handling of mixed algorithms in encrypted digest hash-cipher (generation) and cipher-hash (verification) requests (e.g. SNOW3G + ZUC or ZUC + AES CTR) in QAT PMD possible when running on GEN3 QAT hardware. Such algorithm combinations are not supported on GEN1/GEN2 hardware and executing the request returns `RTE_CRYPTO_OP_STATUS_INVALID_SESSION`.

- **Queue-pairs are now thread-safe on Intel QuickAssist Technology (QAT) PMD.**

  Queue-pairs are thread-safe on Intel CPUs but Queues are not (that is, within a single queue-pair all enqueues to the TX queue must be done from one thread and all dequeues from the RX queue must be done from one thread, but enqueues and dequeues may be done in different threads.).

- **Updated the ZUC PMD.**

  - Transitioned underlying library from libSSO ZUC to intel-ipsec-mb library (minimum version required 0.53).
  - Removed dynamic library limitation, so PMD can be built as a shared object now.

- **Updated the KASUMI PMD.**

  - Transitioned underlying library from libSSO KASUMI to intel-ipsec-mb library (minimum version required 0.53).

- **Updated the SNOW3G PMD.**

  - Transitioned underlying library from libSSO SNOW3G to intel-ipsec-mb library (minimum version required 0.53).

- **Changed armv8 crypto PMD external dependency.**

  Changed armv8 crypto PMD external dependency. The armv8 crypto PMD now depends on the Arm crypto library, and Marvell's armv8 crypto library is not used anymore. The library name has been changed from armv8_crypto to AArch64crypto.

- **Added inline IPsec support to Marvell OCTEON TX2 PMD.**

  Added inline IPsec support to Marvell OCTEON TX2 PMD. With this feature, applications will be able to offload entire IPsec offload to the hardware. For the configured sessions, hardware will do

the lookup and perform decryption and IPsec transformation. For the outbound path, applications can submit a plain packet to the PMD, and it will be sent out on the wire after doing encryption and IPsec transformation of the packet.

- **Added Marvell OCTEON TX2 End Point rawdev PMD.**

  Added a new OCTEON TX2 rawdev PMD for End Point mode of operation. See the *Marvell OCTEON TX2 End Point Rawdev Driver* for more details on this new PMD.

- **Added event mode to l3fwd sample application.**

  Added event device support for the `l3fwd` sample application. It demonstrates usage of poll and event mode IO mechanism under a single application.

- **Added cycle-count mode to the compression performance tool.**

  Enhanced the compression performance tool by adding a cycle-count mode which can be used to help measure and tune hardware and software PMDs.

- **Added OpenWrt howto guide.**

  Added document which describes how to enable DPDK on OpenWrt in both virtual and physical machines.

19.2.2 Removed Items

- **Disabled building all the Linux kernel modules by default.**

  In order to remove the build time dependency on the Linux kernel, the Technical Board decided to disable all the kernel modules by default from 20.02 version.

- **Removed coalescing feature from Intel QuickAssist Technology (QAT) PMD.**

  The internal tail write coalescing feature was removed as not compatible with dual-thread feature. It was replaced with a threshold feature. At busy times if only a small number of packets can be enqueued, each enqueue causes an expensive MMIO write. These MMIO write occurrences can be optimized by using the new threshold parameter on process start. Please see QAT documentation for more details.

19.2.3 API Changes

- No change in this release.

19.2.4 ABI Changes

- No change, kept ABI v20. DPDK 20.02 is compatible with DPDK 19.11.
- The soname for each stable ABI version should be just the ABI version major number without the minor number. Unfortunately both major and minor were used in the v19.11 release, causing version v20.x releases to be incompatible with ABI v20.0.

  The commit f26c2b39b271 fixed the issue by switching from 2-part to 3-part ABI version numbers so that we can keep v20.0 as soname and using the final digits to identify the DPDK 20.x releases which are ABI compatible.

19.2.5 Tested Platforms

- Intel® platforms with Intel® NICs combinations
  - CPU
    - Intel® Atom™ CPU C3758 @ 2.20GHz
    - Intel® Atom™ CPU C3858 @ 2.00GHz
    - Intel® Atom™ CPU C3958 @ 2.00GHz
    - Intel® Xeon® CPU D-1541 @ 2.10GHz
    - Intel® Xeon® CPU D-1553N @ 2.30GHz
    - Intel® Xeon® CPU E5-2680 0 @ 2.70GHz
    - Intel® Xeon® CPU E5-2680 v2 @ 2.80GHz
    - Intel® Xeon® CPU E5-2699 v3 @ 2.30GHz
    - Intel® Xeon® CPU E5-2699 v4 @ 2.20GHz
    - Intel® Xeon® Gold 6139 CPU @ 2.30GHz
    - Intel® Xeon® Gold 6252N CPU @ 2.30GHz
    - Intel® Xeon® Platinum 8180 CPU @ 2.50GHz
    - Intel® Xeon® Platinum 8280M CPU @ 2.70GHz
  - OS:
    - CentOS 7.7
    - CentOS 8.0
    - Fedora 31
    - FreeBSD 12.1
    - Red Hat Enterprise Linux Server release 8.0
    - Red Hat Enterprise Linux Server release 7.7
    - Suse15SP1
    - Ubuntu 14.04
    - Ubuntu 16.04
    - Ubuntu 16.10
    - Ubuntu 18.04
    - Ubuntu 19.04
  - NICs:
    - Intel® Corporation Ethernet Controller E810-C for SFP (4x25G)
      - Firmware version: 1.02 0x80002b69
      - Device id (pf): 8086:1593
      - Driver version: 0.12.34 (ice)

- Intel® Corporation Ethernet Controller E810-C for SFP (2x100G)
  - Firmware version: 1.02 0x80002b68
  - Device id (pf): 8086:1592
  - Driver version: 0.12.34 (ice)
- Intel® 82599ES 10 Gigabit Ethernet Controller
  - Firmware version: 0x61bf0001
  - Device id (pf/vf): 8086:10fb / 8086:10ed
  - Driver version: 5.6.1 (ixgbe)
- Intel® Corporation Ethernet Connection X552/X557-AT 10GBASE-T
  - Firmware version: 0x800003e7
  - Device id (pf/vf): 8086:15ad / 8086:15a8
  - Driver version: 5.1.0 (ixgbe)
- Intel® Corporation Ethernet Controller 10G X550T
  - Firmware version: 0x80000482
  - Device id (pf): 8086:1563
  - Driver version: 5.6.1 (ixgbe)
- Intel® Ethernet Converged Network Adapter X710-DA4 (4x10G)
  - Firmware version: 7.20 0x800079e8
  - Device id (pf/vf): 8086:1572 / 8086:154c
  - Driver version: 2.10.19.30 (i40e)
- Intel® Corporation Ethernet Connection X722 for 10GbE SFP+ (4x10G)
  - Firmware version: 4.11 0x80001def
  - Device id (pf/vf): 8086:37d0 / 8086:37cd
  - Driver version: 2.10.19.30 (i40e)
- Intel® Corporation Ethernet Connection X722 for 10GBASE-T (2x10G)
  - Firmware version: 4.10 0x80001a7a
  - Device id (pf/vf): 8086:37d2 / 8086:37cd
  - Driver version: 2.10.19.30 (i40e)
- Intel® Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
  - Firmware version: 7.20 0x80007947
  - Device id (pf/vf): 8086:158b / 8086:154c
  - Driver version: 2.10.19.30 (i40e)
- Intel® Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
  - Firmware version: 7.20 0x80007948

            - Device id (pf/vf): 8086:1583 / 8086:154c
            - Driver version: 2.10.19.30 (i40e)
        * Intel® Corporation I350 Gigabit Network Connection
            - Firmware version: 1.63, 0x80000cbc
            - Device id (pf/vf): 8086:1521 / 8086:1520
            - Driver version: 5.4.0-k (igb)
        * Intel® Corporation I210 Gigabit Network Connection
            - Firmware version: 3.25, 0x800006eb
            - Device id (pf): 8086:1533
            - Driver version: 5.4.0-k(igb)
- Intel® platforms with Mellanox® NICs combinations
    - CPU:
        * Intel® Xeon® Gold 6154 CPU @ 3.00GHz
        * Intel® Xeon® CPU E5-2697A v4 @ 2.60GHz
        * Intel® Xeon® CPU E5-2697 v3 @ 2.60GHz
        * Intel® Xeon® CPU E5-2680 v2 @ 2.80GHz
        * Intel® Xeon® CPU E5-2650 v4 @ 2.20GHz
        * Intel® Xeon® CPU E5-2640 @ 2.50GHz
        * Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz
    - OS: * Red Hat Enterprise Linux Server release 7.5 (Maipo) * Red Hat Enterprise Linux Server release 7.4 (Maipo) * Red Hat Enterprise Linux Server release 7.3 (Maipo) * Red Hat Enterprise Linux Server release 7.2 (Maipo) * Ubuntu 18.04 * Ubuntu 16.04
    - OFED:
        * MLNX_OFED 4.7-3.2.9.0
        * MLNX_OFED 5.0-0.4.1.0 and above
    - upstream kernel:
        * Linux 5.5 and above
    - rdma-core:
        * rdma-core-28.0-1 and above
    - NICs:
        * Mellanox® ConnectX®-3 Pro 40G MCX354A-FCC_Ax (2x40G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1007
            - Firmware version: 2.42.5000
        * Mellanox® ConnectX®-3 Pro 40G MCX354A-FCCT (2x40G)

        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1007
        - Firmware version: 2.42.5000
      * Mellanox® ConnectX®-4 Lx 25G MCX4121A-ACAT (2x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1015
        - Firmware version: 14.27.1000 and above
      * Mellanox® ConnectX®-4 Lx 50G MCX4131A-GCAT (1x50G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1015
        - Firmware version: 14.27.1000 and above
      * Mellanox® ConnectX®-5 100G MCX516A-CCAT (2x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1017
        - Firmware version: 16.27.1000 and above
      * Mellanox® ConnectX®-5 100G MCX556A-ECAT (2x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1017
        - Firmware version: 16.27.1000 and above
      * Mellanox® ConnectX®-5 100G MCX556A-EDAT (2x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1017
        - Firmware version: 16.27.1000 and above
      * Mellanox® ConnectX®-5 Ex EN 100G MCX516A-CDAT (2x100G)
        - Host interface: PCI Express 4.0 x16
        - Device ID: 15b3:1019
        - Firmware version: 16.27.1000 and above
- Mellanox® BlueField SmartNIC
  - Mellanox® BlueField SmartNIC MT416842 (2x25G)
    * Host interface: PCI Express 3.0 x16
    * Device ID: 15b3:a2d2
    * Firmware version: 18.27.1000
  - SoC Arm cores running OS:
    * CentOS Linux release 7.5.1804 (AltArch)

        * MLNX_OFED 5.0-0.4.0.0
    - DPDK application running on Arm cores inside SmartNIC
- IBM Power 9 platforms with Mellanox® NICs combinations
    - CPU:
        * POWER9 2.2 (pvr 004e 1202) 2300MHz
    - OS:
        * Ubuntu 18.04.1 LTS (Bionic Beaver)
    - NICs:
        * Mellanox® ConnectX®-5 100G MCX556A-ECAT (2x100G)
            · Host interface: PCI Express 3.0 x16
            · Device ID: 15b3:1017
            · Firmware version: 16.27.1000
    - OFED:
        * MLNX_OFED 5.0-0.4.1.0
- ARMv8 SoC combinations from Marvell (with integrated NICs)
    - SoC:
        * CN83xx, CN96xx, CN93xx
    - OS (Based on Marvell OCTEON TX SDK-10.3.2.x):
        * Arch Linux
        * Buildroot 2018.11
        * Ubuntu 16.04.1 LTS
        * Ubuntu 16.10
        * Ubuntu 18.04.1
        * Ubuntu 19.04

19.3 DPDK Release 19.11

19.3.1 New Features

- **Added support for –base-virtaddr EAL option to FreeBSD.**

  The FreeBSD version of DPDK now also supports setting base virtual address for mapping pages and resources into its address space.

- **Added Lock-free Stack for aarch64.**

  Enabled the lock-free stack implementation for aarch64 platforms.

- **Extended pktmbuf mempool private structure.**

  rte_pktmbuf_pool_private structure was extended to include flags field for future compatibility. As per 19.11 release this field is reserved and should be set to 0 by the user.

+* **Changed mempool allocation behavior.**

  Changed the mempool allocation behaviour so that objects no longer cross pages by default. Note, this may consume more memory when using small memory pages.

- **Added support for dynamic fields and flags in mbuf.**

  This new feature adds the ability to dynamically register some room for a field or a flag in the mbuf structure. This is typically used for specific offload features, where adding a static field or flag in the mbuf is not justified.

- **Added support for hairpin queues.**

  On supported NICs, we can now setup hairpin queues which will offload packets from the wire, back to the wire.

- **Added flow tag in rte_flow.**

  The `SET_TAG` action and `TAG` item have been added to support transient flow tag.

- **Extended metadata support in rte_flow.**

  Flow metadata has been extended to both Rx and Tx.

  - Tx metadata can also be set by SET_META action of rte_flow.
  - Rx metadata is delivered to the host via a dynamic field of `rte_mbuf` with `PKT_RX_DYNF_METADATA`.

- **Added ethdev API to set supported packet types.**

  - Added new API `rte_eth_dev_set_ptypes` which allows an application to inform a PMD about a reduced range of packet types to handle.
  - This scheme will allow PMDs to avoid lookup of internal ptype table on Rx and thereby improve Rx performance if the application wishes to do so.

- **Added Rx offload flag to enable or disable RSS update.**

  - Added new Rx offload flag `DEV_RX_OFFLOAD_RSS_HASH` which can be used to enable/disable PMDs write to `rte_mbuf::hash::rss`.
  - PMDs notify the validity of `rte_mbuf::hash:rss` to the application by enabling `PKT_RX_RSS_HASH` flag in `rte_mbuf::ol_flags`.

- **Added Rx/Tx packet burst mode "get" API.**

  Added two new functions `rte_eth_rx_burst_mode_get` and `rte_eth_tx_burst_mode_get` that allow an application to retrieve the mode information about Rx/Tx packet burst such as Scalar or Vector, and Vector technology like AVX2.

- **Added Hisilicon hns3 PMD.**

  Added the new `hns3` net driver for the inbuilt Hisilicon Network Subsystem 3 (HNS3) network engine found in the Hisilicon Kunpeng 920 SoC. See the *HNS3 Poll Mode Driver* guide for more details on this new driver.

- **Added NXP PFE PMD.**

  Added the new PFE driver for the NXP LS1012A platform. See the *PFE Poll Mode Driver* NIC driver guide for more details on this new driver.

- **Updated Broadcom bnxt driver.**

  Updated Broadcom bnxt driver with new features and improvements, including:

  - Added support for hot firmware upgrade.
  - Added support for error recovery.
  - Added support for querying and using COS classification in hardware.
  - Added LRO support Thor devices.
  - Update HWRM API to version 1.10.1.6

- **Updated the enic driver.**

  - Added support for Geneve with options offload.
  - Added flow API implementation based on VIC Flow Manager API.

- **Updated iavf PMD.**

  Enable AVX2 data path for iavf PMD.

- **Updated the Intel e1000 driver.**

  Added support for the `RTE_ETH_DEV_CLOSE_REMOVE` flag.

- **Updated the Intel ixgbe driver.**

  Added support for the `RTE_ETH_DEV_CLOSE_REMOVE` flag.

- **Updated the Intel i40e driver.**

  Added support for the `RTE_ETH_DEV_CLOSE_REMOVE` flag.

- **Updated the Intel fm10k driver.**

  Added support for the `RTE_ETH_DEV_CLOSE_REMOVE` flag.

- **Updated the Intel ice driver.**

  Updated the Intel ice driver with new features and improvements, including:

  - Added support for device-specific DDP package loading.
  - Added support for handling Receive Flex Descriptor.
  - Added support for protocol extraction on per Rx queue.
  - Added support for Flow Director filter based on generic filter framework.
  - Added support for the `RTE_ETH_DEV_CLOSE_REMOVE` flag.
  - Generic filter enhancement - Supported pipeline mode. - Supported new packet type like PPPoE for switch filter.
  - Supported input set change and symmetric hash by rte_flow RSS action.
  - Added support for GTP Tx checksum offload.
  - Added new device IDs to support E810_XXV devices.

- **Updated the Huawei hinic driver.**

  Updated the Huawei hinic driver with new features and improvements, including:

  - Enabled SR-IOV - Partially supported at this point, VFIO only.
  - Supported VLAN filter and VLAN offload.
  - Supported Unicast MAC filter and Multicast MAC filter.
  - Supported Flow API for LACP, VRRP, BGP and so on.
  - Supported FW version get.

- **Updated Mellanox mlx5 driver.**

  Updated Mellanox mlx5 driver with new features and improvements, including:

  - Added support for VLAN pop flow offload command.
  - Added support for VLAN push flow offload command.
  - Added support for VLAN set PCP offload command.
  - Added support for VLAN set VID offload command.
  - Added support for matching on packets withe Geneve tunnel header.
  - Added hairpin support.
  - Added ConnectX-6 Dx support.
  - Flow engine selected based on RDMA Core library version. DV flow engine selected if version is rdma-core-24.0 or higher. Verbs flow engine selected otherwise.

- **Updated the AF_XDP PMD.**

  Updated the AF_XDP PMD. The new features include:

  - Enabled zero copy between application mempools and UMEM by enabling the `XDP_UMEM_UNALIGNED_CHUNKS UMEM` flag.

- **Added cryptodev asymmetric session-less operation.**

  Added a session-less option to the cryptodev asymmetric structure. It works the same way as symmetric crypto, and the corresponding transform is used directly by the crypto operation.

- **Added Marvell NITROX symmetric crypto PMD.**

  Added a symmetric crypto PMD for Marvell NITROX V security processor. See the *Marvell NITROX Crypto Poll Mode Driver* guide for more details on this new PMD.

- **Added asymmetric support to Marvell OCTEON TX crypto PMD.**

  Added support for asymmetric operations to Marvell OCTEON TX crypto PMD. Supports RSA and modexp operations.

- **Added Marvell OCTEON TX2 crypto PMD.**

  Added a new PMD driver for hardware crypto offload block on `OCTEON TX2` SoC.

  See *Marvell OCTEON TX2 Crypto Poll Mode Driver* for more details

- **Updated NXP crypto PMDs for PDCP support.**

Added PDCP support to the DPAA_SEC and DPAA2_SEC PMDs using rte_security APIs. Support has been added for all sequence number sizes for control and user plane. Test and test-crypto-perf applications have been updated for unit testing.

- **Updated the AESNI-MB PMD.**
  - Added support for intel-ipsec-mb version 0.53.
- **Updated the AESNI-GCM PMD.**
  - Added support for intel-ipsec-mb version 0.53.
  - Added support for in-place chained mbufs with AES-GCM algorithm.
- **Enabled Single Pass GCM acceleration on QAT GEN3.**

  Added support for Single Pass GCM, available on QAT GEN3 only (Intel QuickAssist Technology P5xxx). It is automatically chosen instead of the classic 2-pass mode when running on QAT GEN3, significantly improving the performance of AES GCM operations.
- **Updated the Intel QuickAssist Technology (QAT) asymmetric crypto PMD.**
  - Added support for asymmetric session-less operations.
  - Added support for RSA algorithm with pair `(n, d)` private key representation.
  - Added support for RSA algorithm with quintuple private key representation.
- **Updated the Intel QuickAssist Technology (QAT) compression PMD.**

  Added stateful decompression support in the Intel QuickAssist Technology PMD. Please note that stateful compression is not supported.
- **Added external buffers support for dpdk-test-compress-perf tool.**

  Added a command line option to the `dpdk-test-compress-perf` tool to allocate and use memory zones as external buffers instead of keeping the data directly in mbuf areas.
- **Updated the IPSec library.**
  - Added Security Associations (SA) Database API to `librte_ipsec`. A new test-sad application has also been introduced to evaluate and perform custom functional and performance tests for an IPsec SAD implementation.
  - Support fragmented packets in inline crypto processing mode with fallback `lookaside-none` session. Corresponding changes are also added in the IPsec Security Gateway application.
- **Introduced FIFO for NTB PMD.**

  Introduced FIFO for NTB (Non-transparent Bridge) PMD to support packet based processing.
- **Added eBPF JIT support for arm64.**

  Added eBPF JIT support for arm64 architecture to improve the eBPF program performance.
- **Added RIB and FIB (Routing/Forwarding Information Base) libraries.**

  Added Routing and Forwarding Information Base (RIB/FIB) libraries. RIB and FIB can replace the LPM (Longest Prefix Match) library with better control plane (RIB) performance. The data plane (FIB) can be extended with new algorithms.
- **Updated testpmd with a command for ptypes.**

  - Added a console command to testpmd app, `show port (port_id) ptypes` which gives ability to print port supported ptypes in different protocol layers.
  - Packet type detection disabled by default for the supported PMDs.

- **Added new l2fwd-event sample application.**

  Added an example application `l2fwd-event` that adds event device support to the traditional l2fwd example. It demonstrates usage of poll and event mode IO mechanism under a single application.

- **Added build support for Link Time Optimization.**

  LTO is an optimization technique used by the compiler to perform whole program analysis and optimization at link time. In order to do that compilers store their internal representation of the source code that the linker uses at the final stage of the compilation process.

  See *Link Time Optimization* for more information:

- **Added IOVA as VA support for KNI.**
  - Added IOVA = VA support for KNI. KNI can operate in IOVA = VA mode when `iova-mode=va` EAL option is passed to the application or when bus IOVA scheme is selected as RTE_IOVA_VA. This mode only works on Linux Kernel versions 4.10.0 and above.
  - Due to IOVA to KVA address translations, based on the KNI use case there can be a performance impact. For mitigation, forcing IOVA to PA via EAL `--iova-mode=pa` option can be used, IOVA_DC bus iommu scheme can also result in IOVA as PA.

19.3.2 Removed Items

- Removed library-level ABI versions. These have been replaced with a single project-level ABI version for non-experimental libraries and an ABI version of `0` for experimental libraries. Review the *ABI Policy* and *ABI Versioning* guides for more information.
- Removed duplicated set of commands for Rx offload configuration from testpmd:

```
port config all crc-strip|scatter|rx-cksum|rx-timestamp|
               hw-vlan|hw-vlan-filter|hw-vlan-strip|hw-vlan-extend on|off
```

  The testpmd command set that can be used instead in order to enable or disable Rx offloading on all Rx queues of a port is:

```
port config <port_id> rx_offload crc_strip|scatter|
                                 ipv4_cksum|udp_cksum|tcp_cksum|timestamp|
                                 vlan_strip|vlan_filter|vlan_extend on|off
```

- Removed AF_XDP pmd_zero copy vdev argument. Support is now auto-detected.
- The following sample applications have been removed in this release:
  - Exception Path
  - L3 Forwarding in a Virtualization Environment
  - Load Balancer
  - Netmap Compatibility
  - Quota and Watermark
  - vhost-scsi

- Removed arm64-dpaa2-* build config. arm64-dpaa-* can now build for both dpaa and dpaa2 platforms.

19.3.3 API Changes

- eal: made the `lcore_config` struct and global symbol private.
- eal: removed the `rte_cpu_check_supported` function, replaced by `rte_cpu_is_supported` since dpdk v17.08.
- eal: removed the `rte_malloc_virt2phy` function, replaced by `rte_malloc_virt2iova` since v17.11.
- eal: made the `rte_config` struct and `rte_eal_get_configuration` function private.
- mem: hid the internal `malloc_heap` structure and the `rte_malloc_heap.h` header.
- vfio: removed `rte_vfio_dma_map` and `rte_vfio_dma_unmap` that have been marked as deprecated in release 19.05. `rte_vfio_container_dma_map` and `rte_vfio_container_dma_unmap` can be used as substitutes.
- pci: removed the following functions deprecated since dpdk v17.11:
  - `eal_parse_pci_BDF` replaced by `rte_pci_addr_parse`
  - `eal_parse_pci_DomBDF` replaced by `rte_pci_addr_parse`
  - `rte_eal_compare_pci_addr` replaced by `rte_pci_addr_cmp`
- The network structure `esp_tail` has been prefixed by `rte_`.
- The network definitions of PPPoE ethertypes have been prefixed by `RTE_`.
- The network structure for MPLS has been prefixed by `rte_`.
- ethdev: changed `rte_eth_dev_infos_get` return value from `void` to `int` to provide a way to report various error conditions.
- ethdev: changed `rte_eth_promiscuous_enable` and `rte_eth_promiscuous_disable` return value from `void` to `int` to provide a way to report various error conditions.
- ethdev: changed `rte_eth_allmulticast_enable` and `rte_eth_allmulticast_disable` return value from `void` to `int` to provide a way to report various error conditions.
- ethdev: changed `rte_eth_dev_xstats_reset` return value from `void` to `int` to provide a way to report various error conditions.
- ethdev: changed `rte_eth_link_get` and `rte_eth_link_get_nowait` return value from `void` to `int` to provide a way to report various error conditions.
- ethdev: changed `rte_eth_macaddr_get` return value from `void` to `int` to provide a way to report various error conditions.
- ethdev: changed `rte_eth_dev_owner_delete` return value from `void` to `int` to provide a way to report various error conditions.
- ethdev: The deprecated function `rte_eth_dev_count` was removed. The function `rte_eth_dev_count_avail` is a drop-in replacement. If the intent is to iterate over ports, `RTE_ETH_FOREACH_*` macros are better port iterators.

- ethdev: `RTE_FLOW_ITEM_TYPE_META` data endianness altered to host one. Due to the new dynamic metadata field in mbuf is host-endian either, there is a minor compatibility issue for applications in case of 32-bit values supported.
- ethdev: the tx_metadata mbuf field is moved to dynamic one. `PKT_TX_METADATA` flag is replaced with `PKT_TX_DYNF_METADATA`. `DEV_TX_OFFLOAD_MATCH_METADATA` offload flag is removed, now metadata support in PMD is engaged on dynamic field registration.
- event: The function `rte_event_eth_tx_adapter_enqueue` takes an additional input as `flags`. Flag `RTE_EVENT_ETH_TX_ADAPTER_ENQUEUE_SAME_DEST` which has been introduced in this release is used when all the packets enqueued in the Tx adapter are destined for the same Ethernet port and Tx queue.
- sched: The pipe nodes configuration parameters such as number of pipes, pipe queue sizes, pipe profiles, etc., are moved from port level structure to subport level. This allows different subports of the same port to have different configuration for the pipe nodes.

19.3.4 ABI Changes

- policy: Please note the revisions to the *ABI Policy* introducing major ABI versions, with DPDK 19.11 becoming the first major version `v20`. ABI changes to add new features continue to be permitted in subsequent releases, with the condition that ABI compatibility with the major ABI version is maintained.
- net: The Ethernet address and other header definitions have changed attributes. They have been modified to be aligned on 2-byte boundaries. These changes should not impact normal usage because drivers naturally align the Ethernet header on receive and all known encapsulations preserve the alignment of the header.
- security: The field `replay_win_sz` has been moved from the ipsec library based `rte_ipsec_sa_prm` structure to security library based structure `rte_security_ipsec_xform`, which specify the anti-replay window size to enable sequence replay attack handling.
- ipsec: The field `replay_win_sz` has been removed from the structure `rte_ipsec_sa_prm` as it has been added to the security library.
- ethdev: Added 32-bit fields for maximum LRO aggregated packet size, in struct `rte_eth_dev_info` for the port capability and in struct `rte_eth_rxmode` for the port configuration. Application should use the new field in struct `rte_eth_rxmode` to configure the requested size. PMD should use the new field in struct `rte_eth_dev_info` to report the supported port capability.

19.3.5 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
librte_acl.so.2
librte_bbdev.so.1
librte_bitratestats.so.2
librte_bpf.so.1
librte_bus_dpaa.so.2
librte_bus_fslmc.so.2
librte_bus_ifpga.so.2
librte_bus_pci.so.2
```

(continues on next page)

(continued from previous page)

```
  librte_bus_vdev.so.2
  librte_bus_vmbus.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_compressdev.so.1
  librte_cryptodev.so.8
  librte_distributor.so.1
+ librte_eal.so.12
  librte_efd.so.1
+ librte_ethdev.so.13
+ librte_eventdev.so.8
+ librte_fib.so.1
  librte_flow_classify.so.1
  librte_gro.so.1
  librte_gso.so.1
  librte_hash.so.2
  librte_ip_frag.so.1
+ librte_ipsec.so.2
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_latencystats.so.1
  librte_lpm.so.2
  librte_mbuf.so.5
  librte_member.so.1
  librte_mempool.so.5
  librte_meter.so.3
  librte_metrics.so.1
  librte_net.so.1
+ librte_pci.so.2
  librte_pdump.so.3
  librte_pipeline.so.3
  librte_pmd_bnxt.so.2
  librte_pmd_bond.so.2
  librte_pmd_i40e.so.2
  librte_pmd_ixgbe.so.2
  librte_pmd_dpaa2_qdma.so.1
  librte_pmd_ring.so.2
  librte_pmd_softnic.so.1
  librte_pmd_vhost.so.2
  librte_port.so.3
  librte_power.so.1
  librte_rawdev.so.1
+ librte_rib.so.1
  librte_rcu.so.1
  librte_reorder.so.1
  librte_ring.so.2
+ librte_sched.so.4
+ librte_security.so.3
  librte_stack.so.1
  librte_table.so.3
  librte_timer.so.1
  librte_vhost.so.4
```

19.3.6 Known Issues

19.3.7 Tested Platforms

- Intel® platforms with Intel® NICs combinations
  - CPU
    - Intel® Atom™ CPU C3758 @ 2.20GHz
    - Intel® Atom™ CPU C3858 @ 2.00GHz
    - Intel® Atom™ CPU C3958 @ 2.00GHz
    - Intel® Xeon® CPU D-1541 @ 2.10GHz
    - Intel® Xeon® CPU D-1553N @ 2.30GHz
    - Intel® Xeon® CPU E5-2680 0 @ 2.70GHz
    - Intel® Xeon® CPU E5-2680 v2 @ 2.80GHz
    - Intel® Xeon® CPU E5-2699 v3 @ 2.30GHz
    - Intel® Xeon® CPU E5-2699 v4 @ 2.20GHz
    - Intel® Xeon® Gold 6139 CPU @ 2.30GHz
    - Intel® Xeon® Gold 6252N CPU @ 2.30GHz
    - Intel® Xeon® Platinum 8180 CPU @ 2.50GHz
    - Intel® Xeon® Platinum 8280M CPU @ 2.70GHz
  - OS:
    - CentOS 7.6
    - Fedora 30
    - FreeBSD 12.0
    - Red Hat Enterprise Linux Server release 8.0
    - Red Hat Enterprise Linux Server release 7.6
    - Suse12SP3
    - Ubuntu 14.04
    - Ubuntu 16.04
    - Ubuntu 16.10
    - Ubuntu 18.04
    - Ubuntu 19.04
  - NICs:
    - Intel® Corporation Ethernet Controller E810-C for SFP (2x25G)
      - Firmware version: 1.02 0x80002084 1.2538.0/1.02 0x80002082 1.2538.0
      - Device id (pf): 8086:1593
      - Driver version: 0.12.25 (ice)

- Intel® Corporation Ethernet Controller E810-C for SFP (2x100G)
  - Firmware version: 1.02 0x80002081 1.2538.0
  - Device id (pf): 8086:1592
  - Driver version: 0.12.25 (ice)
- Intel® 82599ES 10 Gigabit Ethernet Controller
  - Firmware version: 0x61bf0001
  - Device id (pf/vf): 8086:10fb / 8086:10ed
  - Driver version: 5.6.1 (ixgbe)
- Intel® Corporation Ethernet Connection X552/X557-AT 10GBASE-T
  - Firmware version: 0x800003e7
  - Device id (pf/vf): 8086:15ad / 8086:15a8
  - Driver version: 5.1.0 (ixgbe)
- Intel® Corporation Ethernet Controller 10G X550T
  - Firmware version: 0x80000482
  - Device id (pf): 8086:1563
  - Driver version: 5.6.1 (ixgbe)
- Intel® Ethernet Converged Network Adapter X710-DA4 (4x10G)
  - Firmware version: 7.00 0x80004cdb
  - Device id (pf/vf): 8086:1572 / 8086:154c
  - Driver version: 2.9.21 (i40e)
- Intel® Corporation Ethernet Connection X722 for 10GbE SFP+ (4x10G)
  - Firmware version: 4.10 0x80001a3c
  - Device id (pf/vf): 8086:37d0 / 8086:37cd
  - Driver version: 2.9.21 (i40e)
- Intel® Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
  - Firmware version: 7.00 0x80004cf8
  - Device id (pf/vf): 8086:158b / 8086:154c
  - Driver version: 2.9.21 (i40e)
- Intel® Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
  - Firmware version: 7.00 0x80004c97
  - Device id (pf/vf): 8086:1583 / 8086:154c
  - Driver version: 2.9.21 (i40e)
- Intel® Corporation I350 Gigabit Network Connection
  - Firmware version: 1.63, 0x80000cbc

            - Device id (pf/vf): 8086:1521 / 8086:1520
            - Driver version: 5.4.0-k (igb)
        * Intel® Corporation I210 Gigabit Network Connection
            - Firmware version: 3.25, 0x800006eb
            - Device id (pf): 8086:1533
            - Driver version: 5.4.0-k(igb)
- ARMv8 SoC combinations from Marvell (with integrated NICs)
  - SoC:
    * CN83xx, CN96xx, CN93xx
  - OS (Based on Marvell OCTEON TX SDK-10.1.2.0):
    * Arch Linux
    * Buildroot 2018.11
    * Ubuntu 16.04.1 LTS
    * Ubuntu 16.10
    * Ubuntu 18.04.1
    * Ubuntu 19.04
- Intel® platforms with Mellanox® NICs combinations
  - CPU:
    * Intel® Xeon® Gold 6154 CPU @ 3.00GHz
    * Intel® Xeon® CPU E5-2697A v4 @ 2.60GHz
    * Intel® Xeon® CPU E5-2697 v3 @ 2.60GHz
    * Intel® Xeon® CPU E5-2680 v2 @ 2.80GHz
    * Intel® Xeon® CPU E5-2650 v4 @ 2.20GHz
    * Intel® Xeon® CPU E5-2640 @ 2.50GHz
    * Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz
  - OS:
    * Red Hat Enterprise Linux Server release 8.0 (Maipo)
    * Red Hat Enterprise Linux Server release 7.7 (Maipo)
    * Red Hat Enterprise Linux Server release 7.6 (Maipo)
    * Red Hat Enterprise Linux Server release 7.5 (Maipo)
    * Red Hat Enterprise Linux Server release 7.4 (Maipo)
    * Red Hat Enterprise Linux Server release 7.3 (Maipo)
    * Red Hat Enterprise Linux Server release 7.2 (Maipo)
    * Ubuntu 19.04

    - Ubuntu 18.10
    - Ubuntu 18.04
    - Ubuntu 16.04
    - SUSE Linux Enterprise Server 15
  - OFED:
    - MLNX_OFED 4.6-1.0.1.1
    - MLNX_OFED 4.7-1.0.0.1
    - MLNX_OFED 4.7-3.1.9.0 and above
  - upstream kernel:
    - Linux 5.3 and above
  - rdma-core:
    - rdma-core-24.1-1 and above
  - NICs:
    - Mellanox® ConnectX®-3 Pro 40G MCX354A-FCC_Ax (2x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1007
      - Firmware version: 2.42.5000
    - Mellanox® ConnectX®-4 10G MCX4111A-XCAT (1x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.26.2032 and above
    - Mellanox® ConnectX®-4 10G MCX4121A-XCAT (2x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.26.2032 and above
    - Mellanox® ConnectX®-4 25G MCX4111A-ACAT (1x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.26.2032 and above
    - Mellanox® ConnectX®-4 25G MCX4121A-ACAT (2x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.26.2032 and above
    - Mellanox® ConnectX®-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)

- Host interface: PCI Express 3.0 x8
- Device ID: 15b3:1013
- Firmware version: 12.26.2032 and above

* Mellanox® ConnectX®-4 40G MCX415A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.26.2032 and above
* Mellanox® ConnectX®-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.26.2032 and above
* Mellanox® ConnectX®-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.26.2032 and above
* Mellanox® ConnectX®-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.26.2032 and above
  - Firmware version: 12.26.2032 and above
* Mellanox® ConnectX®-4 50G MCX415A-CCAT (1x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.26.2032 and above
* Mellanox® ConnectX®-4 100G MCX416A-CCAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.26.2032 and above
* Mellanox® ConnectX®-4 Lx 10G MCX4121A-XCAT (2x10G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.26.2032 and above
* Mellanox® ConnectX®-4 Lx 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8

            - Device ID: 15b3:1015
            - Firmware version: 14.26.2032 and above
        * Mellanox® ConnectX®-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.26.2032 and above
        * Mellanox® ConnectX®-5 Ex EN 100G MCX516A-CDAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1019
            - Firmware version: 16.26.2032 and above
- IBM Power 9 platforms with Mellanox® NICs combinations
    - CPU:
        * POWER9 2.2 (pvr 004e 1202) 2300MHz
    - OS:
        * Ubuntu 18.04.1 LTS (Bionic Beaver)
    - NICs:
        * Mellanox® ConnectX®-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.26.1040
    - OFED:
        * MLNX_OFED 4.7-1.0.0.2

19.4 DPDK Release 19.08

19.4.1 New Features

- **EAL will now pick IOVA as VA mode as the default in most cases.**

  Previously, the preferred default IOVA mode was selected to be IOVA as PA. The behavior has now been changed to handle IOVA mode detection in a more complex manner, and will default to IOVA as VA in most cases.

- **Added MCS lock.**

  MCS lock provides scalability by spinning on a CPU/thread local variable which avoids expensive cache bouncing. It provides fairness by maintaining a list of acquirers and passing the lock to each CPU/thread in the order they acquired the lock.

- **Updated the EAL Pseudo-random Number Generator.**

  The `lrand48()` based `rte_rand()` function is replaced with a DPDK-native combined Linear Feedback Shift Register (LFSR) pseudo-random number generator (PRNG).

  This new PRNG implementation is multi-thread safe, provides higher-quality pseudo-random numbers (including full 64 bit support) and improved performance.

  In addition, `<rte_random.h>` is extended with a new function `rte_rand_max()` which supplies unbiased, bounded pseudo-random numbers.

- **Updated the Broadcom bnxt PMD.**

  Updated the Broadcom bnxt PMD. The major enhancements include:

  - Performance optimizations in non-vector Tx path.
  - Added support for SSE vector mode.
  - Updated HWRM API to version 1.10.0.91.

- **Added support for Broadcom NetXtreme-E BCM57500 Ethernet controllers.**

  Added support to the Broadcom bnxt PMD for the BCM57500 (a.k.a. "Thor") family of Ethernet controllers. These controllers support link speeds up to 200Gbps, 50G PAM-4, and PCIe 4.0.

- **Added Huawei hinic PMD.**

  Added the new `hinic` net driver for Huawei Intelligent PCIE Network Adapters based on the Huawei Ethernet Controller Hi1822. See the *HINIC Poll Mode Driver* guide for more details on this new driver.

- **Updated the Intel ice driver.**

  Updated the Intel ice driver with new features and improvements, including:

  - Enabled Tx outer/inner L3/L4 checksum offload.
  - Enabled generic filter framework and supported switch filter.
  - Supported UDP tunnel port add.

- **Updated the Intel i40e driver.**

  Updated tje Intel i40e driver with new features and improvements, including:

  - Added support for MARK + RSS action in rte_flow (non-vector RX path only)

- **Updated Mellanox mlx5 driver.**

  Updated Mellanox mlx5 driver with new features and improvements, including:

  - Updated the packet header modification feature. Added support of TCP header sequence number and acknowledgment number modification.
  - Added support for match on ICMP/ICMP6 code and type.
  - Added support for matching on GRE's key and C,K,S present bits.
  - Added support for IP-in-IP tunnel.
  - Accelerated flows with count action creation and destroy.
  - Accelerated flows counter query.
  - Improved Tx datapath performance with enabled HW offloads.

  - Added support for LRO.

- **Updated Solarflare network PMD.**

  Updated the Solarflare `sfc_efx` driver with changes including:

  - Added support for Rx interrupts.

- **Added memif PMD.**

  Added a new Shared Memory Packet Interface (`memif`) PMD. See the *Memif Poll Mode Driver* guide for more details on this new driver.

- **Updated the AF_XDP PMD.**

  Updated the AF_XDP PMD. The new features include:

  - Enabled zero copy through mbuf's external memory mechanism to achieve higher performance.
  - Added multi-queue support to allow one af_xdp vdev with multiple netdev queues.
  - Enabled "need_wakeup" feature which can provide efficient support for the usecase where the application and driver executing on the same core.

- **Enabled infinite Rx in the PCAP PMD.**

  Added an infinite Rx feature to the PCAP PMD which allows packets in the Rx PCAP to be received repeatedly at a high rate. This can be useful for quick performance testing of DPDK apps.

- **Enabled receiving no packet in the PCAP PMD.**

  Added function to allow users to run the PCAP PMD without receiving any packets on PCAP Rx. When the function is called, a dummy queue is created for each Tx queue argument passed.

- **Added a FPGA_LTE_FEC bbdev PMD.**

  Added a new `fpga_lte_fec` bbdev driver for the Intel® FPGA PAC (Programmable Acceleration Card) N3000. See the *Intel(R) FPGA LTE FEC Poll Mode Driver* BBDEV guide for more details on this new driver.

- **Updated the TURBO_SW bbdev PMD.**

  Updated the `turbo_sw` bbdev driver with changes including:

  - Added option to build the driver with or without dependency of external SDK libraries.
  - Added support for 5GNR encode/decode operations.

- **Updated the Intel QuickAssist Technology (QAT) symmetric crypto PMD.**

  Added support for digest-encrypted cases where digest is appended to the data.

- **Added the Intel QuickData Technology PMD.**

  Added a PMD for the Intel® QuickData Technology, part of Intel® I/O Acceleration Technology (Intel I/OAT), which allows data copies to be done by hardware instead of via software, reducing cycles spent copying large blocks of data in applications.

- **Added Marvell OCTEON TX2 drivers.**

  Added the new `ethdev`, `eventdev`, `mempool`, `eventdev Rx adapter`, `eventdev Tx adapter`, `eventdev Timer adapter` and `rawdev DMA` drivers for various HW co-processors available in `OCTEON TX2` SoC.

See *Marvell OCTEON TX2 Platform Guide* and driver information:

  - *OCTEON TX2 Poll Mode driver*
  - *OCTEON TX2 NPA Mempool Driver*
  - *OCTEON TX2 SSO Eventdev Driver*
  - *OCTEON TX2 DMA Driver*

- **Introduced the Intel NTB PMD.**

  Added a PMD for Intel NTB (Non-transparent Bridge). This PMD implements a handshake between two separate hosts and can share local memory for peer host to directly access.

- **Updated the IPSec library and IPsec Security Gateway application.**

  Added the following features to `librte_ipsec`. Corresponding changes are also added in the `ipsec-secgw` sample application.

  - ECN and DSCP field header reconstruction as per RFC4301.
  - Transport mode with IPv6 extension headers.
  - Support packets with multiple segments.

- **Updated telemetry library for global metrics support.**

  Updated `librte_telemetry` to fetch the global metrics from the `librte_metrics` library.

- **Added new telemetry mode for l3fwd-power application.**

  Added a telemetry mode to the `l3fwd-power` application to report application level busyness, empty and full polls of `rte_eth_rx_burst()`.

- **Updated the pdump application.**

  Add support for pdump to exit with primary process.

- **Updated test-compress-perf tool application.**

  Added a multiple cores feature to the compression perf tool application.

19.4.2 Removed Items

- Removed KNI ethtool, `CONFIG_RTE_KNI_KMOD_ETHTOOL`, support.
- build: armv8 crypto extension is disabled.

19.4.3 API Changes

- The `rte_mem_config` structure has been made private. New accessor `rte_mcfg_*` functions were introduced to provide replacement for direct access to the shared mem config.
- The network structures, definitions and functions have been prefixed by `rte_` to resolve conflicts with libc headers.
- malloc: The function `rte_malloc_set_limit()` was never implemented. It is deprecated and will be removed in a future release.

- cryptodev: the `uint8_t *data` member of the `key` structure in the xforms structure (`rte_crypto_cipher_xform`, `rte_crypto_auth_xform`, and `rte_crypto_aead_xform`) have been changed to `const uint8_t *data`.
- eventdev: No longer marked as experimental.

  The eventdev functions are no longer marked as experimental, and have become part of the normal DPDK API and ABI. Any future ABI changes will be announced at least one release before the ABI change is made. There are no ABI breaking changes planned.
- ip_frag: The IP fragmentation library converts input mbuf into fragments using input MTU size via the `rte_ipv4_fragment_packet()` interface. Once fragmentation is done, each `mbuf->ol_flags` are set to enable IP checksum H/W offload irrespective of the platform capability. Cleared IP checksum H/W offload flag from the library. The application must set this flag if it is supported by the platform and application wishes to use it.
- ip_frag: IP reassembly library converts the list of fragments into a reassembled packet via `rte_ipv4_frag_reassemble_packet()` interface. Once reassembly is done, `mbuf->ol_flags` are set to enable IP checksum H/W offload irrespective of the platform capability. Cleared IP checksum H/W offload flag from the library. The application must set this flag if it is supported by the platform and application wishes to use it.
- sched: Macros `RTE_SCHED_QUEUES_PER_TRAFFIC_CLASS` and `RTE_SCHED_PIPE_PROFILES_PER_PORT` are removed for flexible configuration of pipe traffic classes and their queues size, and for runtime configuration of the maximum number of pipe profiles, respectively. In addition, the `wrr_weights` field of struct `rte_sched_pipe_params` is modified to be used only for best-effort tc, and the `qsize` field of struct `rte_sched_port_params` is changed to allow different sizes for each queue.

19.4.4 ABI Changes

- eventdev: Event based Rx adapter callback

  The mbuf pointer array in the event eth Rx adapter callback has been replaced with an event array. Using an event array allows the application to change attributes of the events enqueued by the SW adapter.

  The callback can drop packets and populate a callback argument with the number of dropped packets. Add a Rx adapter stats field to keep track of the total number of dropped packets.
- cryptodev: New member in `rte_cryptodev_config` to allow applications to disable features supported by the crypto device. Only the following features would be allowed to be disabled this way,
  - `RTE_CRYPTODEV_FF_SYMMETRIC_CRYPTO`.
  - `RTE_CRYPTODEV_FF_ASYMMETRIC_CRYPTO`.
  - `RTE_CRYPTODEV_FF_SECURITY`.

  Disabling unused features would facilitate efficient usage of HW/SW offload.
- bbdev: New operations and parameters have been added to support new 5GNR operations. The bbdev ABI is still kept experimental.
- rawdev: The driver names have been changed to `librte_rawdev_*`. Now they all have the same prefix, and same name with make and meson builds.

19.4.5 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_bbdev.so.1
  librte_bitratestats.so.2
  librte_bpf.so.1
  librte_bus_dpaa.so.2
  librte_bus_fslmc.so.2
  librte_bus_ifpga.so.2
  librte_bus_pci.so.2
  librte_bus_vdev.so.2
  librte_bus_vmbus.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_compressdev.so.1
+ librte_cryptodev.so.8
  librte_distributor.so.1
+ librte_eal.so.11
  librte_efd.so.1
  librte_ethdev.so.12
+ librte_eventdev.so.7
  librte_flow_classify.so.1
  librte_gro.so.1
  librte_gso.so.1
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_ipsec.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_latencystats.so.1
  librte_lpm.so.2
  librte_mbuf.so.5
  librte_member.so.1
  librte_mempool.so.5
  librte_meter.so.3
  librte_metrics.so.1
  librte_net.so.1
  librte_pci.so.1
  librte_pdump.so.3
  librte_pipeline.so.3
  librte_pmd_bnxt.so.2
  librte_pmd_bond.so.2
  librte_pmd_i40e.so.2
  librte_pmd_ixgbe.so.2
  librte_pmd_dpaa2_qdma.so.1
  librte_pmd_ring.so.2
  librte_pmd_softnic.so.1
  librte_pmd_vhost.so.2
  librte_port.so.3
  librte_power.so.1
  librte_rawdev.so.1
  librte_rcu.so.1
  librte_reorder.so.1
  librte_ring.so.2
+ librte_sched.so.3
  librte_security.so.2
  librte_stack.so.1
  librte_table.so.3
  librte_timer.so.1
```

(continues on next page)

(continued from previous page)

```
librte_vhost.so.4
```

19.4.6 Known Issues

- **Unsuitable IOVA mode may be picked as the default.**

  Not all kernel drivers and not all devices support all IOVA modes. EAL will attempt to pick a reasonable default based on a number of factors, but there may be cases where the default is unsuitable.

  It is recommended to use the *–iova-mode* command-line parameter if the default is not suitable.

19.4.7 Tested Platforms

- Intel(R) platforms with Intel(R) NICs combinations
  - CPU
    - Intel(R) Atom(TM) CPU C3758 @ 2.20GHz
    - Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
    - Intel(R) Xeon(R) CPU D-1553N @ 2.30GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
    - Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
    - Intel(R) Xeon(R) Gold 6139 CPU @ 2.30GHz
    - Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz
    - Intel(R) Xeon(R) Platinum 8280M CPU @ 2.70GHz
  - OS:
    - CentOS 7.6
    - Fedora 30
    - FreeBSD 12.0
    - Red Hat Enterprise Linux Server release 8.0
    - Red Hat Enterprise Linux Server release 7.6
    - Suse12SP3
    - Ubuntu 16.04
    - Ubuntu 16.10
    - Ubuntu 18.04
    - Ubuntu 19.04
  - NICs:
    - Intel(R) 82599ES 10 Gigabit Ethernet Controller

- Firmware version: 0x61bf0001
- Device id (pf/vf): 8086:10fb / 8086:10ed
- Driver version: 5.6.1 (ixgbe)

* Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
  - Firmware version: 0x800003e7
  - Device id (pf/vf): 8086:15ad / 8086:15a8
  - Driver version: 5.1.0 (ixgbe)
* Intel Corporation Ethernet Controller 10G X550T
  - Firmware version: 0x80000482
  - Device id (pf): 8086:1563
  - Driver version: 5.6.1 (ixgbe)
* Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
  - Firmware version: 7.00 0x80004cdb
  - Device id (pf/vf): 8086:1572 / 8086:154c
  - Driver version: 2.9.21 (i40e)
* Intel(R) Corporation Ethernet Connection X722 for 10GbE SFP+ (4x10G)
  - Firmware version: 4.10 0x80001a3c
  - Device id (pf/vf): 8086:37d0 / 8086:37cd
  - Driver version: 2.9.21 (i40e)
* Intel(R) Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
  - Firmware version: 7.00 0x80004cf8
  - Device id (pf/vf): 8086:158b / 8086:154c
  - Driver version: 2.9.21 (i40e)
* Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
  - Firmware version: 7.00 0x80004c97
  - Device id (pf/vf): 8086:1583 / 8086:154c
  - Driver version: 2.9.21 (i40e)
* Intel(R) Corporation I350 Gigabit Network Connection
  - Firmware version: 1.63, 0x80000cbc
  - Device id (pf/vf): 8086:1521 / 8086:1520
  - Driver version: 5.4.0-k (igb)
* Intel Corporation I210 Gigabit Network Connection
  - Firmware version: 3.25, 0x800006eb
  - Device id (pf): 8086:1533

            - Driver version: 5.4.0-k(igb)

- Intel(R) platforms with Mellanox(R) NICs combinations
  - CPU:
    - Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz
    - Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
    - Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
    - Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
    - Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
  - OS:
    - Red Hat Enterprise Linux Server release 7.6 (Maipo)
    - Red Hat Enterprise Linux Server release 7.5 (Maipo)
    - Red Hat Enterprise Linux Server release 7.4 (Maipo)
    - Red Hat Enterprise Linux Server release 7.3 (Maipo)
    - Red Hat Enterprise Linux Server release 7.2 (Maipo)
    - Ubuntu 19.04
    - Ubuntu 18.10
    - Ubuntu 18.04
    - Ubuntu 16.04
    - SUSE Linux Enterprise Server 15
  - OFED:
    - MLNX_OFED 4.6-1.0.1.1
    - MLNX_OFED 4.6-4.1.2.0
  - NICs:
    - Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1007
      - Firmware version: 2.42.5000
    - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.25.6406 and above
    - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)

- Host interface: PCI Express 3.0 x8
- Device ID: 15b3:1013
- Firmware version: 12.25.6406 and above

* Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.6406 and above
* Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.6406 and above
* Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.6406 and above
* Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.25.6406 and above
* Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.6406 and above
* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.6406 and above
* Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.25.6406 and above
  - Firmware version: 12.25.6406 and above
* Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
  - Host interface: PCI Express 3.0 x16

            - Device ID: 15b3:1013
            - Firmware version: 12.25.6406 and above
        * Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1013
            - Firmware version: 12.25.6406 and above
        * Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.25.6406 and above
        * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.25.6406 and above
        * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.25.6406 and above
        * Mellanox(R) ConnectX(R)-5 Ex EN 100G MCX516A-CDAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1019
            - Firmware version: 16.25.6406 and above

- Mellanox(R) BlueField SmartNIC
    - Mellanox(R) BlueField SmartNIC MT416842 (2x25G)
        * Host interface: PCI Express 3.0 x16
        * Device ID: 15b3:a2d2
        * Firmware version: 18.25.6600
    - SoC Arm cores running OS:
        * CentOS Linux release 7.5.1804 (AltArch)
        * MLNX_OFED 4.6-3.5.8.0
    - DPDK application running on Arm cores inside SmartNIC
- IBM Power 9 platforms with Mellanox(R) NICs combinations
    - CPU:
        * POWER9 2.2 (pvr 004e 1202) 2300MHz

  - OS:
    - Ubuntu 18.04.1 LTS (Bionic Beaver)
  - NICs:
    - Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
      - Host interface: PCI Express 3.0 x16
      - Device ID: 15b3:1017
      - Firmware version: 16.25.6406
  - OFED:
    - MLNX_OFED 4.6-4.1.2.0
- ARMv8 SoC combinations from Marvell (with integrated NICs)
  - SoC:
    - CN83xx, CN96xx, CNF95xx, CN93xx
  - OS (Based on Marvell OCTEON TX SDK 10.0):
    - Arch Linux
    - Buildroot 2018.11
    - Ubuntu 16.04.1 LTS
    - Ubuntu 16.10
    - Ubuntu 18.04.1
    - Ubuntu 19.04

19.5 DPDK Release 19.05

19.5.1 New Features

- **Added new armv8 machine targets.**

  Added new armv8 machine targets:

  - BlueField (Mellanox)
  - OcteonTX2 (Marvell)
  - ThunderX2 (Marvell)
- **Added Windows Support.**

  Added Windows support to build Hello World sample application.
- **Added Stack Library.**

  Added a new stack library and APIs for configuration and use of a bounded stack of pointers. The API provides multi-thread safe push and pop operations that can operate on one or more pointers per operation.

The library supports two stack implementations: standard (lock-based) and lock-free. The lock-free implementation is currently limited to x86-64 platforms.

- **Added Lock-Free Stack Mempool Handler.**

  Added a new lock-free stack handler, which uses the newly added stack library.

- **Added RCU library.**

  Added RCU library supporting a quiescent state based memory reclamation method. This library helps identify the quiescent state of the reader threads so that the writers can free the memory associated with the lock free data structures.

- **Updated KNI module and PMD.**

  Updated the KNI kernel module to set the `max_mtu` according to the given initial MTU size. Without it, the maximum MTU was 1500.

  Updated the KNI PMD driver to set the `mbuf_size` and MTU based on the given mb-pool. This provide the ability to pass jumbo frames if the mb-pool contains a suitable buffer size.

- **Added the AF_XDP PMD.**

  Added a Linux-specific PMD driver for AF_XDP. This PMD can create an AF_XDP socket and bind it to a specific netdev queue. It allows a DPDK application to send and receive raw packets through the socket which would bypass the kernel network stack to achieve high performance packet processing.

- **Added a net PMD NFB.**

  Added the new `nfb` net driver for Netcope NFB cards. See the *NFB poll mode driver library* NIC guide for more details on this new driver.

- **Added IPN3KE net PMD.**

  Added the new `ipn3ke` net driver for the Intel® FPGA PAC (Programmable Acceleration Card) N3000. See the *IPN3KE Poll Mode Driver* NIC guide for more details on this new driver.

  In addition `ifpga_rawdev` was also updated to support Intel® FPGA PAC N3000 with SPI interface access, I2C Read/Write, and Ethernet PHY configuration.

- **Updated Solarflare network PMD.**

  Updated the Solarflare `sfc_efx` driver with changes including:

  - Added support for Rx descriptor status and related API in a secondary process.
  - Added support for Tx descriptor status API in a secondary process.
  - Added support for RSS RETA and hash configuration reading API in a secondary process.
  - Added support for Rx packet types list in a secondary process.
  - Added Tx prepare to do Tx offloads checks.
  - Added support for VXLAN and GENEVE encapsulated TSO.

- **Updated Mellanox mlx4 driver.**

  Updated Mellanox mlx4 driver with new features and improvements, including:

  - Added firmware version reading.
  - Added support for secondary processes.

  - Added support of per-process device registers. Reserving identical VA space is not needed anymore.
  - Added support for multicast address list interfaces.

- **Updated Mellanox mlx5 driver.**

  Updated Mellanox mlx5 driver with new features and improvements, including:

  - Added firmware version reading.
  - Added support for new naming scheme of representor.
  - Added support for new PCI device DMA map/unmap API.
  - Added support for multiport InfiniBand device.
  - Added control of excessive memory pinning by kernel.
  - Added support of DMA memory registration by secondary process.
  - Added support of per-process device registers. Reserving identical VA space is not required anymore.
  - Added support for jump action for both E-Switch and NIC.
  - Added Support for multiple rte_flow groups in NIC steering.
  - **Flow engine re-designed to support large scale deployments. this includes:**
    * Support millions of offloaded flow rules.
    * Fast flow insertion and deletion up to 1M flow update per second.

- **Renamed avf to iavf.**

  Renamed Intel Ethernet Adaptive Virtual Function driver `avf` to `iavf`, which includes the directory name, lib name, filenames, makefile, docs, macros, functions, structs and any other strings in the code.

- **Updated the enic driver.**

  Updated enic driver with new features and improvements, including:

  - Fixed several flow (director) bugs related to MARK, SCTP, VLAN, VXLAN, and inner packet matching.
  - Added limited support for RAW.
  - Added limited support for RSS.
  - Added limited support for PASSTHRU.

- **Updated the ixgbe driver.**

  Updated the ixgbe driver to add promiscuous mode support for the VF.

- **Updated the ice driver.**

  Updated ice driver with new features and improvements, including:

  - Added support of SSE and AVX2 instructions in Rx and Tx paths.
  - Added package download support.
  - Added Safe Mode support.

  - Supported RSS for UPD/TCP/SCTP+IPV4/IPV6 packets.

- **Updated the i40e driver.**

  New features for PF in the i40e driver:

  - Added support for VXLAN-GPE packet.
  - Added support for VXLAN-GPE classification.

- **Updated the ENETC driver.**

  Updated ENETC driver with new features and improvements, including:

  - Added physical addressing mode support.
  - Added SXGMII interface support.
  - Added basic statistics support.
  - Added promiscuous and allmulticast mode support.
  - Added MTU update support.
  - Added jumbo frame support.
  - Added queue start/stop.
  - Added CRC offload support.
  - Added Rx checksum offload validation support.

- **Updated the atlantic PMD.**

  Added MACSEC hardware offload experimental API.

- **Updated the Intel QuickAssist Technology (QAT) compression PMD.**

  Updated the Intel QuickAssist Technology (QAT) compression PMD to simplify, and make more robust, the handling of Scatter Gather Lists (SGLs) with more than 16 segments.

- **Updated the QuickAssist Technology (QAT) symmetric crypto PMD.**

  Added support for AES-XTS with 128 and 256 bit AES keys.

- **Added Intel QuickAssist Technology PMD for asymmetric crypto.**

  Added a new QAT Crypto PMD which provides asymmetric cryptography algorithms. Modular exponentiation and modular multiplicative inverse algorithms were added in this release.

- **Updated AESNI-MB PMD.**

  Added support for out-of-place operations.

- **Updated the IPsec library.**

  The IPsec library has been updated with AES-CTR and 3DES-CBC cipher algorithms support. The related `ipsec-secgw` test scripts have been added.

- **Updated the testpmd application.**

  Improved the `testpmd` application performance on ARM platform. For `macswap` forwarding mode, NEON intrinsics are now used to do swap to save CPU cycles.

- **Updated power management library.**

  Added support for Intel Speed Select Technology - Base Frequency (SST-BF). The `rte_power_get_capabilities` struct now has a bit in it's returned mask indicating if it is a high frequency core.

- **Updated distributor sample application.**

  Added support for the Intel SST-BF feature so that the distributor core is pinned to a high frequency core if available.

19.5.2 API Changes

- eal: the type of the `attr_value` parameter of the function `rte_service_attr_get()` has been changed from `uint32_t *` to `uint64_t *`.
- meter: replace `enum rte_meter_color` in the meter library with new `rte_color` definition added in 19.02. Replacements with `rte_color` values has been performed in many places such as `rte_mtr.h` and `rte_tm.h` to consolidate multiple color definitions.
- vfio: Functions `rte_vfio_container_dma_map` and `rte_vfio_container_dma_unmap` have been extended with an option to request mapping or un-mapping to the default vfio container fd.
- power: `rte_power_set_env` and `rte_power_unset_env` functions have been modified to be thread safe.
- timer: Functions have been introduced that allow multiple instances of the timer lists to be created. In addition they are now allocated in shared memory. New functions allow particular timer lists to be selected when timers are being started, stopped, and managed.

19.5.3 ABI Changes

- ethdev: Additional fields in rte_eth_dev_info.

  The `rte_eth_dev_info` structure has had two extra fields added: `min_mtu` and `max_mtu`. Each of these are of type `uint16_t`. The values of these fields can be set specifically by the PMD drivers as supported values can vary from device to device.

- cryptodev: in 18.08 a new structure `rte_crypto_asym_op` was introduced and included into `rte_crypto_op`. As the `rte_crypto_asym_op` structure was defined as cache-line aligned that caused unintended changes in `rte_crypto_op` structure layout and alignment. Remove cache-line alignment for `rte_crypto_asym_op` to restore expected `rte_crypto_op` layout and alignment.
- timer: `rte_timer_subsystem_init` now returns success or failure to reflect whether it was able to allocate memory.

19.5.4 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_bbdev.so.1
  librte_bitratestats.so.2
  librte_bpf.so.1
  librte_bus_dpaa.so.2
  librte_bus_fslmc.so.2
  librte_bus_ifpga.so.2
  librte_bus_pci.so.2
  librte_bus_vdev.so.2
  librte_bus_vmbus.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_compressdev.so.1
+ librte_cryptodev.so.7
  librte_distributor.so.1
+ librte_eal.so.10
  librte_efd.so.1
+ librte_ethdev.so.12
  librte_eventdev.so.6
  librte_flow_classify.so.1
  librte_gro.so.1
  librte_gso.so.1
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_ipsec.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_latencystats.so.1
  librte_lpm.so.2
  librte_mbuf.so.5
  librte_member.so.1
  librte_mempool.so.5
  librte_meter.so.3
  librte_metrics.so.1
  librte_net.so.1
  librte_pci.so.1
  librte_pdump.so.3
  librte_pipeline.so.3
  librte_pmd_bnxt.so.2
  librte_pmd_bond.so.2
  librte_pmd_i40e.so.2
  librte_pmd_ixgbe.so.2
  librte_pmd_dpaa2_qdma.so.1
  librte_pmd_ring.so.2
  librte_pmd_softnic.so.1
  librte_pmd_vhost.so.2
  librte_port.so.3
  librte_power.so.1
  librte_rawdev.so.1
+ librte_rcu.so.1
  librte_reorder.so.1
  librte_ring.so.2
  librte_sched.so.2
  librte_security.so.2
+ librte_stack.so.1
  librte_table.so.3
  librte_timer.so.1
```

(continues on next page)

(continued from previous page)

```
librte_vhost.so.4
```

19.5.5 Known Issues

- **On x86 platforms, AVX512 support is disabled with binutils 2.31.**

  Due to a defect in binutils 2.31 AVX512 support is disabled. DPDK defect: https://bugs.dpdk.org/show_bug.cgi?id=249 GCC defect: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90028

- **No software AES-XTS implementation.**

  There are currently no cryptodev software PMDs available which implement support for the AES-XTS algorithm, so this feature can only be used if compatible hardware and an associated PMD is available.

19.5.6 Tested Platforms

- Intel(R) platforms with Intel(R) NICs combinations
  - CPU
    - Intel(R) Atom(TM) CPU C3758 @ 2.20GHz
    - Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
    - Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
    - Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz
    - Intel(R) Xeon(R) Gold 6139 CPU @ 2.30GHz
  - OS:
    - CentOS 7.4
    - CentOS 7.5
    - Fedora 25
    - Fedora 28
    - Fedora 29
    - FreeBSD 12.0
    - Red Hat Enterprise Linux Server release 7.4
    - Red Hat Enterprise Linux Server release 7.5
    - Red Hat Enterprise Linux Server release 7.6
    - SUSE12SP3
    - Open SUSE 15
    - Wind River Linux 8

    * Ubuntu 14.04
    * Ubuntu 16.04
    * Ubuntu 16.10
    * Ubuntu 18.04
    * Ubuntu 18.10
  - NICs:
    * Intel(R) 82599ES 10 Gigabit Ethernet Controller
      - Firmware version: 0x61bf0001
      - Device id (pf/vf): 8086:10fb / 8086:10ed
      - Driver version: 5.2.3 (ixgbe)
    * Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      - Firmware version: 0x800003e7
      - Device id (pf/vf): 8086:15ad / 8086:15a8
      - Driver version: 4.4.6 (ixgbe)
    * Intel Corporation Ethernet Controller 10G X550T
      - Firmware version: 0x80000482
      - Device id (pf): 8086:1563
      - Driver version: 5.1.0-k(ixgbe)
    * Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
      - Firmware version: 6.80 0x80003cc1
      - Device id (pf/vf): 8086:1572 / 8086:154c
      - Driver version: 2.7.29 (i40e)
    * Intel(R) Corporation Ethernet Connection X722 for 10GbE SFP+ (4x10G)
      - Firmware version: 3.33 0x80000fd5 0.0.0
      - Device id (pf/vf): 8086:37d0 / 8086:37cd
      - Driver version: 2.7.29 (i40e)
    * Intel(R) Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
      - Firmware version: 6.80 0x80003d05
      - Device id (pf/vf): 8086:158b / 8086:154c
      - Driver version: 2.7.29 (i40e)
    * Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
      - Firmware version: 6.80 0x80003cfb
      - Device id (pf/vf): 8086:1583 / 8086:154c
      - Driver version: 2.7.29 (i40e)

        * Intel(R) Corporation I350 Gigabit Network Connection
            - Firmware version: 1.63, 0x80000dda
            - Device id (pf/vf): 8086:1521 / 8086:1520
            - Driver version: 5.4.0-k (igb)
        * Intel Corporation I210 Gigabit Network Connection
            - Firmware version: 3.25, 0x800006eb, 1.1824.0
            - Device id (pf): 8086:1533
            - Driver version: 5.4.0-k(igb)
- Intel(R) platforms with Mellanox(R) NICs combinations
    - CPU:
        * Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz
        * Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
        * Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
        * Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
        * Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
        * Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
        * Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
    - OS:
        * Red Hat Enterprise Linux Server release 7.6 (Maipo)
        * Red Hat Enterprise Linux Server release 7.5 (Maipo)
        * Red Hat Enterprise Linux Server release 7.4 (Maipo)
        * Red Hat Enterprise Linux Server release 7.3 (Maipo)
        * Red Hat Enterprise Linux Server release 7.2 (Maipo)
        * Ubuntu 19.04
        * Ubuntu 18.10
        * Ubuntu 18.04
        * Ubuntu 16.04
        * SUSE Linux Enterprise Server 15
    - MLNX_OFED: 4.5-1.0.1.0
    - MLNX_OFED: 4.6-1.0.1.1
    - NICs:
        * Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1007

- Firmware version: 2.42.5000

* Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.1020 and above
* Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.1020 and above
* Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.1020 and above
* Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.1020 and above
* Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.1020 and above
* Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.25.1020 and above
* Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.1020 and above
* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.25.1020 and above
* Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)

            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1013
            - Firmware version: 12.25.1020 and above
            - Firmware version: 12.25.1020 and above
        * Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1013
            - Firmware version: 12.25.1020 and above
        * Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1013
            - Firmware version: 12.25.1020 and above
        * Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.25.1020 and above
        * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.25.1020 and above
        * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.25.1020 and above
        * Mellanox(R) ConnectX(R)-5 Ex EN 100G MCX516A-CDAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1019
            - Firmware version: 16.25.1020 and above

- Arm platforms with Mellanox(R) NICs combinations
    - CPU:
        * Qualcomm Arm 1.1 2500MHz
    - OS:
        * Red Hat Enterprise Linux Server release 7.5 (Maipo)
    - NICs:

        * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.24.0220
        * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.24.0220

- Mellanox(R) BlueField SmartNIC
    - Mellanox(R) BlueField SmartNIC MT416842 (2x25G)
        * Host interface: PCI Express 3.0 x16
        * Device ID: 15b3:a2d2
        * Firmware version: 18.25.1010
    - SoC Arm cores running OS:
        * CentOS Linux release 7.4.1708 (AltArch)
        * MLNX_OFED 4.6-1.0.0.0
    - DPDK application running on Arm cores inside SmartNIC
- IBM Power 9 platforms with Mellanox(R) NICs combinations
    - CPU:
        * POWER9 2.2 (pvr 004e 1202) 2300MHz
    - OS:
        * Ubuntu 18.04.1 LTS (Bionic Beaver)
    - NICs:
        * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.24.1000
    - OFED:
        * MLNX_OFED_LINUX-4.6-1.0.1.0

19.6 DPDK Release 19.02

19.6.1 New Features

- **Added support for freeing hugepages exactly as originally allocated.**

  Some applications using memory event callbacks (especially for managing RDMA memory regions) require that memory be freed back to the system exactly as it was originally allocated. These applications typically also require that a malloc allocation not span across two separate hugepage allocations. A new `--match-allocations` EAL init flag has been added to fulfill both of these requirements.

- **Added API to register external memory in DPDK.**

  A new `rte_extmem_register/rte_extmem_unregister` API was added to allow chunks of external memory to be registered with DPDK without adding them to the malloc heap.

- **Added support for using virtio-user without hugepages.**

  The `--no-huge` mode was augmented to use memfd-backed memory (on systems that support memfd), to allow using virtio-user-based NICs without hugepages.

- **Release of the ENA PMD v2.0.0.**

  Version 2.0.0 of the ENA PMD was added with the following additions:

  - Added Low Latency Queue v2 (LLQv2). This feature reduces the latency of the packets by pushing the header directly through the PCI to the device. This allows the NIC to start handle packets right after the doorbell without waiting for DMA.
  - Added independent configuration of HW Tx and Rx ring depths.
  - Added support for up to 8k Rx descriptors per ring.
  - Added additional doorbell check on Tx, to handle Tx more efficiently for big bursts of packets.
  - Added per queue statistics.
  - Added extended statistics using xstats DPDK API.
  - The reset routine was aligned with the DPDK API, so now it can be handled as in other PMDs.
  - Fixed out of order (OOO) completion.
  - Fixed memory leaks due to port stops and starts in the middle of traffic.
  - Updated documentation and features list of the PMD.

- **Updated mlx5 driver.**

  Updated the mlx5 driver including the following changes:

  - Fixed `imissed` counter to be reported through `rte_eth_stats` instead of `rte_eth_xstats`.
  - Added packet header modification through Direct Verbs flow driver.
  - Added ConnectX-6 PCI device ID to be proved by `mlx5` driver.
  - Added flow counter support to Direct Verbs flow driver though DevX.

  - Renamed build options for the glue layer to `CONFIG_RTE_IBVERBS_LINK_DLOPEN` for make and `ibverbs_link` for meson.
  - Added static linkage of `mlx` dependency.
  - Improved stability of E-Switch flow driver.
  - Added new make build configuration to set the cacheline size for BlueField correctly - `arm64-bluefield-linux-gcc`.

- **Updated the enic driver.**
  - Added support for the `RTE_ETH_DEV_CLOSE_REMOVE` flag.
  - Added a handler to get the firmware version string.
  - Added support for multicast filtering.

- **Added dynamic queues allocation support for i40e VF.**

  Previously, the available VF queues were reserved by PF at initialization stage. Now both DPDK PF and Kernel PF (>=2.1.14) will support dynamic queue allocation. At runtime, when VF requests for more queue exceed the initial reserved amount, the PF can allocate up to 16 queues as the request after a VF reset.

- **Added ICE net PMD.**

  Added the new `ice` net driver for Intel(R) Ethernet Network Adapters E810. See the *ICE Poll Mode Driver* NIC guide for more details on this new driver.

- **Added support for SW-assisted VDPA live migration.**

  This SW-assisted VDPA live migration facility helps VDPA devices without logging capability to perform live migration, a mediated SW relay can help devices to track dirty pages caused by DMA. the IFC driver has enabled this SW-assisted live migration mode.

- **Added security checks to the cryptodev symmetric session operations.**

  Added a set of security checks to the access cryptodev symmetric session. The checks include the session's user data read/write check and the session private data referencing status check while freeing a session.

- **Updated the AESNI-MB PMD.**
  - Added support for intel-ipsec-mb version 0.52.
  - Added AES-GMAC algorithm support.
  - Added Plain SHA1, SHA224, SHA256, SHA384, and SHA512 algorithms support.

- **Added IPsec Library.**

  Added an experimental library `librte_ipsec` to provide ESP tunnel and transport support for IPv4 and IPv6 packets.

  The library provides support for AES-CBC ciphering and AES-CBC with HMAC-SHA1 algorithm-chaining, and AES-GCM and NULL algorithms only at present. It is planned to add more algorithms in future releases.

  See *IPsec Packet Processing Library* for more information.

- **Updated the ipsec-secgw sample application.**

The `ipsec-secgw` sample application has been updated to use the new `librte_ipsec` library, which has also been added in this release. The original functionality of ipsec-secgw is retained, a new command line parameter `-l` has been added to ipsec-secgw to use the IPsec library, instead of the existing IPsec code in the application.

The IPsec library does not support all the functionality of the existing ipsec-secgw application. It is planned to add the outstanding functionality in future releases.

See *IPsec Security Gateway Sample Application* for more information.

- **Enabled checksum support in the ISA-L compressdev driver.**

  Added support for both adler and crc32 checksums in the ISA-L PMD. This aids data integrity across both compression and decompression.

- **Added a compression performance test tool.**

  Added a new performance test tool to test the compressdev PMD. The tool tests compression ratio and compression throughput.

- **Added intel_pstate support to Power Management library.**

  Previously, using the power management library required the disabling of the intel_pstate kernel driver, and the enabling of the acpi_cpufreq kernel driver. This is no longer the case, as the use of the intel_pstate kernel driver is now supported, and automatically detected by the library.

19.6.2 API Changes

- eal: Function `rte_bsf64` in `rte_bitmap.h` has been renamed to `rte_bsf64_safe` and moved to `rte_common.h`. A new `rte_bsf64` function has been added in `rte_common.h` that follows the convention set by the existing `rte_bsf32` function.
- eal: Segment fd API on Linux now sets error code to `ENOTSUP` in more cases where segment the fd API is not expected to be supported:
  - On attempt to get a segment fd for an externally allocated memory segment
  - In cases where memfd support would have been required to provide segment fds (such as in-memory or no-huge mode)
- eal: Functions `rte_malloc_dump_stats()`, `rte_malloc_dump_heaps()` and `rte_malloc_get_socket_stats()` are no longer safe to call concurrently with `rte_malloc_heap_create()` or `rte_malloc_heap_destroy()` function calls.
- mbuf: `RTE_MBUF_INDIRECT()`, which was deprecated in 18.05, was replaced with `RTE_MBUF_CLONED()` and removed in 19.02.
- sched: As result of the new format of the mbuf sched field, the functions `rte_sched_port_pkt_write()` and `rte_sched_port_pkt_read_tree_path()` got an additional parameter of type `struct rte_sched_port`.
- pdump: The `rte_pdump_set_socket_dir()`, the parameter `path` of `rte_pdump_init()` and enum `rte_pdump_socktype` were deprecated since 18.05 and are removed in this release.
- cryptodev: The parameter `session_pool` in the function `rte_cryptodev_queue_pair_setup()` is removed.
- cryptodev: a new function `rte_cryptodev_sym_session_pool_create()` has been introduced. This function is now mandatory when creating symmetric session header mempool. Please

note all crypto applications are required to use this function from now on. Failed to do so will cause a `rte_cryptodev_sym_session_create()` function call return error.

19.6.3 ABI Changes

- mbuf: The format of the sched field of `rte_mbuf` has been changed to include the following fields: `queue ID`, `traffic class`, `color`.
- cryptodev: as shown in the 18.11 deprecation notice, the structure `rte_cryptodev_qp_conf` has added two parameters for symmetric session mempool and symmetric session private data mempool.
- cryptodev: as shown in the 18.11 deprecation notice, the structure `rte_cryptodev_sym_session` has been updated to contain more information to ensure safely accessing the session and session private data.
- security: A new field `uint64_t opaque_data` has been added to `rte_security_session` structure. That would allow upper layer to easily associate/de-associate some user defined data with the security session.

19.6.4 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_bbdev.so.1
  librte_bitratestats.so.2
  librte_bpf.so.1
  librte_bus_dpaa.so.2
  librte_bus_fslmc.so.2
  librte_bus_ifpga.so.2
  librte_bus_pci.so.2
  librte_bus_vdev.so.2
  librte_bus_vmbus.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_compressdev.so.1
+ librte_cryptodev.so.6
  librte_distributor.so.1
  librte_eal.so.9
  librte_efd.so.1
  librte_ethdev.so.11
  librte_eventdev.so.6
  librte_flow_classify.so.1
  librte_gro.so.1
  librte_gso.so.1
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_latencystats.so.1
  librte_lpm.so.2
+ librte_mbuf.so.5
  librte_member.so.1
  librte_mempool.so.5
  librte_meter.so.2
```

(continues on next page)

(continued from previous page)

```
  librte_metrics.so.1
  librte_net.so.1
  librte_pci.so.1
+ librte_pdump.so.3
  librte_pipeline.so.3
  librte_pmd_bnxt.so.2
  librte_pmd_bond.so.2
  librte_pmd_i40e.so.2
  librte_pmd_ixgbe.so.2
  librte_pmd_dpaa2_qdma.so.1
  librte_pmd_ring.so.2
  librte_pmd_softnic.so.1
  librte_pmd_vhost.so.2
  librte_port.so.3
  librte_power.so.1
  librte_rawdev.so.1
  librte_reorder.so.1
  librte_ring.so.2
+ librte_sched.so.2
+ librte_security.so.2
  librte_table.so.3
  librte_timer.so.1
  librte_vhost.so.4
```

19.6.5 Known Issues

- AVX-512 support has been disabled for GCC builds when binutils 2.30 is detected [1] because of a crash [2]. This can affect native machine type build targets on the platforms that support AVX512F like Intel Skylake processors, and can cause a possible performance drop. The immediate workaround is to use clang compiler on these platforms. Initial workaround in DPDK v18.11 was to disable AVX-512 support for GCC completely, but based on information on defect submitted to GCC community [3], issue has been identified as binutils 2.30 issue. Since currently only GCC generates AVX-512 instructions, the scope is limited to GCC and binutils 2.30
  - [1]: Commit ("mk: fix scope of disabling AVX512F support")
  - [2]: https://bugs.dpdk.org/show_bug.cgi?id=97
  - [3]: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=88096

19.6.6 Tested Platforms

- Intel(R) platforms with Intel(R) NICs combinations
  - CPU
    - Intel(R) Atom(TM) CPU C3758 @ 2.20GHz
    - Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
    - Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
    - Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz

    - Intel(R) Xeon(R) Gold 6139 CPU @ 2.30GHz
  - OS:
    - CentOS 7.4
    - CentOS 7.5
    - Fedora 25
    - Fedora 28
    - FreeBSD 11.2
    - FreeBSD 12.0
    - Red Hat Enterprise Linux Server release 7.4
    - Red Hat Enterprise Linux Server release 7.5
    - Open SUSE 15
    - Wind River Linux 8
    - Ubuntu 14.04
    - Ubuntu 16.04
    - Ubuntu 16.10
    - Ubuntu 18.04
    - Ubuntu 18.10
  - NICs:
    - Intel(R) 82599ES 10 Gigabit Ethernet Controller
      - Firmware version: 0x61bf0001
      - Device id (pf/vf): 8086:10fb / 8086:10ed
      - Driver version: 5.2.3 (ixgbe)
    - Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      - Firmware version: 0x800003e7
      - Device id (pf/vf): 8086:15ad / 8086:15a8
      - Driver version: 4.4.6 (ixgbe)
    - Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
      - Firmware version: 6.80 0x80003cc1
      - Device id (pf/vf): 8086:1572 / 8086:154c
      - Driver version: 2.7.26 (i40e)
    - Intel(R) Corporation Ethernet Connection X722 for 10GbE SFP+ (4x10G)
      - Firmware version: 3.33 0x80000fd5 0.0.0
      - Device id (pf/vf): 8086:37d0 / 8086:37cd
      - Driver version: 2.7.26 (i40e)

        * Intel(R) Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
            - Firmware version: 6.80 0x80003d05
            - Device id (pf/vf): 8086:158b / 8086:154c
            - Driver version: 2.7.26 (i40e)
        * Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
            - Firmware version: 6.80 0x80003cfb
            - Device id (pf/vf): 8086:1583 / 8086:154c
            - Driver version: 2.7.26 (i40e)
        * Intel(R) Corporation I350 Gigabit Network Connection
            - Firmware version: 1.63, 0x80000dda
            - Device id (pf/vf): 8086:1521 / 8086:1520
            - Driver version: 5.4.0-k (igb)
- Intel(R) platforms with Mellanox(R) NICs combinations
    - CPU:
        * Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz
        * Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
        * Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
        * Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
        * Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
        * Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
        * Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
    - OS:
        * Red Hat Enterprise Linux Server release 7.6 (Maipo)
        * Red Hat Enterprise Linux Server release 7.5 (Maipo)
        * Red Hat Enterprise Linux Server release 7.4 (Maipo)
        * Red Hat Enterprise Linux Server release 7.3 (Maipo)
        * Red Hat Enterprise Linux Server release 7.2 (Maipo)
        * Ubuntu 18.10
        * Ubuntu 18.04
        * Ubuntu 17.10
        * Ubuntu 16.04
        * SUSE Linux Enterprise Server 15
    - MLNX_OFED: 4.4-2.0.1.0
    - MLNX_OFED: 4.5-1.0.1.0

- NICs:
  - Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1007
    - Firmware version: 2.42.5000
  - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.24.1000 and above
  - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.24.1000 and above
  - Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.24.1000 and above
  - Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.24.1000 and above
  - Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.24.1000 and above
  - Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - Firmware version: 12.24.1000 and above
  - Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.24.1000 and above
  - Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)

        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.24.1000 and above
    * Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1013
        - Firmware version: 12.24.1000 and above
        - Firmware version: 12.24.1000 and above
    * Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1013
        - Firmware version: 12.24.1000 and above
    * Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1013
        - Firmware version: 12.24.1000 and above
    * Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1015
        - Firmware version: 14.24.1000 and above
    * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1015
        - Firmware version: 14.24.1000 and above
    * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1017
        - Firmware version: 16.24.1000 and above
    * Mellanox(R) ConnectX(R)-5 Ex EN 100G MCX516A-CDAT (2x100G)
        - Host interface: PCI Express 4.0 x16
        - Device ID: 15b3:1019
        - Firmware version: 16.24.1000 and above

- ARM platforms with Mellanox(R) NICs combinations
  - CPU:

        * Qualcomm ARM 1.1 2500MHz
    - OS:
        * Red Hat Enterprise Linux Server release 7.5 (Maipo)
    - NICs:
        * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
            · Host interface: PCI Express 3.0 x8
            · Device ID: 15b3:1015
            · Firmware version: 14.24.0220
        * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
            · Host interface: PCI Express 3.0 x16
            · Device ID: 15b3:1017
            · Firmware version: 16.24.0220
- Mellanox(R) BlueField SmartNIC
    - Mellanox(R) BlueField SmartNIC MT416842 (2x25G)
        * Host interface: PCI Express 3.0 x16
        * Device ID: 15b3:a2d2
        * Firmware version: 18.24.0328
    - SoC ARM cores running OS:
        * CentOS Linux release 7.4.1708 (AltArch)
        * MLNX_OFED 4.4-2.5.9.0
    - DPDK application running on ARM cores inside SmartNIC
- Power 9 platforms with Mellanox(R) NICs combinations
    - CPU:
        * POWER9 2.2 (pvr 004e 1202) 2300MHz
    - OS:
        * Ubuntu 18.04.1 LTS (Bionic Beaver)
    - NICs:
        * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
            · Host interface: PCI Express 3.0 x16
            · Device ID: 15b3:1017
            · Firmware version: 16.23.1020
    - OFED:
        * MLNX_OFED_LINUX-4.5-1.0.1.0

19.7 DPDK Release 18.11

19.7.1 New Features

- **Added support for using externally allocated memory in DPDK.**

  DPDK has added support for creating new `rte_malloc` heaps referencing memory that was created outside of DPDK's own page allocator, and using that memory natively with any other DPDK library or data structure.

- **Added check for ensuring allocated memory is addressable by devices.**

  Some devices can have addressing limitations so a new function, `rte_mem_check_dma_mask()`, has been added for checking that allocated memory is not out of the device range. Since memory can now be allocated dynamically after initialization, a DMA mask is stored and any new allocated memory will be checked against it and rejected if it is out of range. If more than one device has addressing limitations, the DMA mask is the more restrictive one.

- **Updated the C11 memory model version of the ring library.**

  Added changes to decrease latency for architectures using the C11 memory model version of the ring library.

  On Cavium ThunderX2 platform, the changes decreased latency by 27-29% and 3-15% for MPMC and SPSC cases respectively (with 2 lcores). The real improvements may vary with the number of contending lcores and the size of the ring.

- **Added hot-unplug handle mechanism.**

  Added `rte_dev_hotplug_handle_enable()` and `rte_dev_hotplug_handle_disable()` for enabling or disabling the hotplug handle mechanism.

- **Added support for device multi-process hotplug.**

  Added support for hotplug and hot-unplug in a multiprocessing scenario. Any ethdev devices created in the primary process will be regarded as shared and will be available for all DPDK processes. Synchronization between processes will be done using DPDK IPC.

- **Added new Flow API actions to rewrite fields in packet headers.**

  Added new Flow API actions to:

  - Modify source and destination IP addresses in the outermost IPv4/IPv6 headers.
  - Modify source and destination port numbers in the outermost TCP/UDP headers.

- **Added new Flow API action to swap MAC addresses in Ethernet header.**

  Added new Flow API action to swap the source and destination MAC addresses in the outermost Ethernet header.

- **Add support to offload more flow match and actions for CXGBE PMD.**

  The Flow API support has been enhanced for the CXGBE Poll Mode Driver to offload:

  - Match items: destination MAC address.
  - Action items: push/pop/rewrite vlan header, rewrite IP addresses in outermost IPv4/IPv6 header, rewrite port numbers in outermost TCP/UDP header, swap MAC addresses in outermost Ethernet header.

- **Added a devarg to use the latest supported vector path in i40e.**

  A new devarg `use-latest-supported-vec` was introduced to allow users to choose the latest vector path that the platform supported. For example, users can use AVX2 vector path on BDW/HSW to get better performance.

- **Added support for SR-IOV in netvsc PMD.**

  The `netvsc` poll mode driver now supports the Accelerated Networking SR-IOV option in Hyper-V and Azure. This is an alternative to the previous vdev_netvsc, tap, and failsafe drivers combination.

- **Added a new net driver for Marvell Armada 3k device.**

  Added the new `mvneta` net driver for Marvell Armada 3k device. See the *MVNETA Poll Mode Driver* NIC guide for more details on this new driver.

- **Added NXP ENETC PMD.**

  Added the new enetc driver for the NXP enetc platform. See the *ENETC Poll Mode Driver* NIC driver guide for more details on this new driver.

- **Added Ethernet poll mode driver for Aquantia aQtion family of 10G devices.**

  Added the new `atlantic` ethernet poll mode driver for Aquantia XGBE devices. See the *Aquantia Atlantic DPDK Driver* NIC driver guide for more details on this driver.

- **Updated mlx5 driver.**

  Updated the mlx5 driver including the following changes:

  - Improved security of PMD to prevent the NIC from getting stuck when the application misbehaves.
  - Reworked flow engine to supported e-switch flow rules (transfer attribute).
  - Added support for header re-write(L2-L4), VXLAN encap/decap, count, match on TCP flags and multiple flow groups with e-switch flow rules.
  - Added support for match on metadata, VXLAN and MPLS encap/decap with flow rules.
  - Added support for `RTE_ETH_DEV_CLOSE_REMOVE` flag to provide better support for representors.
  - Added support for meson build.
  - Fixed build issue with PPC.
  - Added support for BlueField VF.
  - Added support for externally allocated static memory for DMA.

- **Updated Solarflare network PMD.**

  Updated the sfc_efx driver including the following changes:

  - Added support for Rx scatter in EF10 datapath implementation.
  - Added support for Rx descriptor status API in EF10 datapath implementation.
  - Added support for TSO in EF10 datapath implementation.
  - Added support for Tx descriptor status API in EF10 (ef10 and ef10_simple) datapaths implementation.

- **Updated the enic driver.**
  - Added AVX2-based vectorized Rx handler.
  - Added VLAN and checksum offloads to the simple Tx handler.
  - Added the "count" flow action.
  - Enabled the virtual address IOVA mode.
- **Updated the failsafe driver.**

  Updated the failsafe driver including the following changes:
  - Added support for Rx and Tx queues start and stop.
  - Added support for Rx and Tx queues deferred start.
  - Added support for runtime Rx and Tx queues setup.
  - Added support multicast MAC address set.
- **Added a devarg to use a PCAP interface physical MAC address.**

  A new devarg `phy_mac` was introduced to allow users to use the physical MAC address of the selected PCAP interface.
- **Added TAP Rx/Tx queues sharing with a secondary process.**

  Added support to allow a secondary process to attach a TAP device created in the primary process, probe the queues, and process Rx/Tx in a secondary process.
- **Added classification and metering support to SoftNIC PMD.**

  Added support for flow classification (rte_flow API), and metering and policing (rte_mtr API) to the SoftNIC PMD.
- **Added Crypto support to the Softnic PMD.**

  The Softnic is now capable of processing symmetric crypto workloads such as cipher, cipher-authentication chaining, and AEAD encryption and decryption. This is achieved by calling DPDK Cryptodev APIs.
- **Added cryptodev port to port library.**

  Cryptodev port is a shim layer in the port library that interacts with DPDK Cryptodev PMDs including burst enqueuing and dequeuing crypto operations.
- **Added symmetric cryptographic actions to the pipeline library.**

  In the pipeline library support was added for symmetric crypto action parsing and an action handler was implemented. The action allows automatic preparation of the crypto operation with the rules specified such as algorithm, key, and IV, etc. for the cryptodev port to process.
- **Updated the AESNI MB PMD.**

  The AESNI MB PMD has been updated with additional support for the AES-GCM algorithm.
- **Added NXP CAAM JR PMD.**

  Added the new caam job ring driver for NXP platforms. See the *NXP CAAM JOB RING (caam_jr)* guide for more details on this new driver.

- **Added support for GEN3 devices to Intel QAT driver.**

  Added support for the third generation of Intel QuickAssist devices.

- **Updated the QAT PMD.**

  The QAT PMD was updated with additional support for:

  - The AES-CMAC algorithm.

- **Added support for Dynamic Huffman Encoding to Intel QAT comp PMD.**

  The Intel QuickAssist (QAT) compression PMD has been updated with support for Dynamic Huffman Encoding for the Deflate algorithm.

- **Added Event Ethernet Tx Adapter.**

  Added event ethernet Tx adapter library that provides configuration and data path APIs for the ethernet transmit stage of an event driven packet processing application. These APIs abstract the implementation of the transmit stage and allow the application to use eventdev PMD support or a common implementation.

- **Added Distributed Software Eventdev PMD.**

  Added the new Distributed Software Event Device (DSW), which is a pure-software eventdev driver distributing the work of scheduling among all eventdev ports and the lcores using them. DSW, compared to the SW eventdev PMD, sacrifices load balancing performance to gain better event scheduling throughput and scalability.

- **Added extendable bucket feature to hash library (rte_hash).**

  This new "extendable bucket" feature provides 100% insertion guarantee to the capacity specified by the user by extending hash table with extra buckets when needed to accommodate the unlikely event of intensive hash collisions. In addition, the internal hashing algorithm was changed to use partial-key hashing to improve memory efficiency and lookup performance.

- **Added lock free reader/writer concurrency to hash library (rte_hash).**

  Lock free reader/writer concurrency prevents the readers from getting blocked due to a preempted writer thread. This allows the hash library to be used in scenarios where the writer thread runs on the control plane.

- **Added Traffic Pattern Aware Power Control Library.**

  Added an experimental library that extends the Power Library and provides empty_poll APIs. This feature measures how many times empty_polls are executed per core and uses the number of empty polls as a hint for system power management.

  See the *Power Management* section of the DPDK Programmers Guide document for more information.

- **Added JSON power policy interface for containers.**

  Extended the Power Library and vm_power_manager sample app to allow power policies to be submitted via a FIFO using JSON formatted strings. Previously limited to Virtual Machines, this feature extends power policy functionality to containers and host applications that need to have their cores frequency controlled based on the rules contained in the policy.

- **Added Telemetry API.**

Added a new telemetry API which allows applications to transparently expose their telemetry in JSON via a UNIX socket. The JSON can be consumed by any Service Assurance agent, such as CollectD.

- **Updated KNI kernel module, rte_kni library, and KNI sample application.**

  Updated the KNI kernel module with a new kernel module parameter, `carrier=[on|off]` to allow the user to control the default carrier state of the KNI kernel network interfaces. The default carrier state is now set to `off`, so the interfaces cannot be used until the carrier state is set to `on` via `rte_kni_update_link` or by writing 1 to `/sys/devices/virtual/net/<iface>/carrier`. In previous versions the default carrier state was left undefined. See *Kernel NIC Interface* for more information.

  Also added the new API function `rte_kni_update_link()` to allow the user to set the carrier state of the KNI kernel network interface.

  Also added a new command line flag `-m` to the KNI sample application to monitor and automatically reflect the physical NIC carrier state to the KNI kernel network interface with the new `rte_kni_update_link()` API. See *Kernel NIC Interface Sample Application* for more information.

- **Added ability to switch queue deferred start flag on testpmd app.**

  Added a console command to testpmd app, giving ability to switch `rx_deferred_start` or `tx_deferred_start` flag of the specified queue of the specified port. The port must be stopped before the command call in order to reconfigure queues.

- **Add a new sample application for vDPA.**

  The vdpa sample application creates vhost-user sockets by using the vDPA backend. vDPA stands for vhost Data Path Acceleration which utilizes virtio ring compatible devices to serve virtio driver directly to enable datapath acceleration. As vDPA driver can help to set up vhost datapath, this application doesn't need to launch dedicated worker threads for vhost enqueue/dequeue operations.

- **Added cryptodev FIPS validation example application.**

  Added an example application to parse and perform symmetric cryptography computation to the NIST Cryptographic Algorithm Validation Program (CAVP) test vectors.

- **Allow unit test binary to take parameters from the environment.**

  The unit test "test", or "dpdk-test", binary is often called from scripts, which can make passing additional parameters, such as a coremask, difficult. Support has been added to the application to allow it to take additional command-line parameter values from the `DPDK_TEST_PARAMS` environment variable to make this application easier to use.

19.7.2 API Changes

- eal: `rte_memseg_list` structure now has an additional flag indicating whether the memseg list is externally allocated. This will have implications for any users of memseg-walk-related functions, as they will now have to skip externally allocated segments in most cases if the intent is to only iterate over internal DPDK memory.

  In addition the `socket_id` parameter across the entire DPDK has gained additional meaning, as some socket ID's will now be representing externally allocated memory. No changes will be required for existing code as backwards compatibility will be kept, and those who do not use this feature will not see these extra socket ID's. Any new API's must not check socket ID parameters

themselves, and must instead leave it to the memory subsystem to decide whether socket ID is a valid one.

- eal: The following devargs functions, which were deprecated in 18.05, were removed in 18.11: `rte_eal_parse_devargs_str()`, `rte_eal_devargs_add()`, `rte_eal_devargs_type_count()`, and `rte_eal_devargs_dump()`.
- eal: The parameters of the function `rte_devargs_remove()` have changed from bus and device names to `struct rte_devargs`.
- eal: The deprecated functions attach/detach were removed in 18.11. `rte_eal_dev_attach` can be replaced by `rte_dev_probe` or `rte_eal_hotplug_add`. `rte_eal_dev_detach` can be replaced by `rte_dev_remove` or `rte_eal_hotplug_remove`.
- eal: The scope of `rte_eal_hotplug_add()`/`rte_dev_probe()` and `rte_eal_hotplug_remove()`/`rte_dev_remove()` has been extended. In the multi-process model, they will guarantee that the device is attached or detached on all processes.
- mbuf: The `__rte_mbuf_raw_free()` and `__rte_pktmbuf_prefree_seg()` functions were deprecated since 17.05 and are replaced by `rte_mbuf_raw_free()` and `rte_pktmbuf_prefree_seg()`.
- ethdev: The deprecated functions attach/detach were removed in 18.11. `rte_eth_dev_attach()` can be replaced by `RTE_ETH_FOREACH_MATCHING_DEV` and `rte_dev_probe()` or `rte_eal_hotplug_add()`. `rte_eth_dev_detach()` can be replaced by `rte_dev_remove()` or `rte_eal_hotplug_remove()`.
- ethdev: A call to `rte_eth_dev_release_port()` has been added in `rte_eth_dev_close()`. As a consequence, a closed port is freed and seen as invalid because of its state `RTE_ETH_DEV_UNUSED`. This new behavior is enabled per driver for a migration period.
- A new device flag, `RTE_ETH_DEV_NOLIVE_MAC_ADDR`, changes the order of actions inside `rte_eth_dev_start()` regarding MAC set. Some NICs do not support MAC changes once the port has started and with this new device flag the MAC can be properly configured in any case. This is particularly important for bonding.
- The default behavior of CRC strip offload has changed in this release. Without any specific Rx offload flag, default behavior by a PMD is now to strip CRC. `DEV_RX_OFFLOAD_CRC_STRIP` offload flag has been removed. To request keeping CRC, application should set `DEV_RX_OFFLOAD_KEEP_CRC` Rx offload.
- eventdev: The type of the second parameter to `rte_event_eth_rx_adapter_caps_get()` has been changed from uint8_t to uint16_t.
- kni: By default, interface carrier status is `off` which means there won't be any traffic. It can be set to `on` via `rte_kni_update_link()` API or via `sysfs` interface: `echo 1 > /sys/class/net/vEth0/carrier`.

  Note interface should be `up` to be able to read/write sysfs interface. When KNI sample application is used, `-m` parameter can be used to automatically update the carrier status for the interface.
- kni: When ethtool support is enabled (`CONFIG_RTE_KNI_KMOD_ETHTOOL=y`) ethtool commands `ETHTOOL_GSET & ETHTOOL_SSET` are no longer supported for kernels that have `ETHTOOL_GLINKSETTINGS & ETHTOOL_SLINKSETTINGS` support. This means `ethtool "-a|--show-pause", "-s|--change"` won't work, and `ethtool <iface>` output will have less information.

19.7.3 ABI Changes

- eal: added `legacy_mem` and `single_file_segments` values to `rte_config` structure on account of improving DPDK usability when using either `--legacy-mem` or `--single-file-segments` flags.
- eal: EAL library ABI version was changed due to previously announced work on supporting external memory in DPDK:
  - Structure `rte_memseg_list` now has a new field indicating length of memory addressed by the segment list
  - Structure `rte_memseg_list` now has a new flag indicating whether the memseg list refers to external memory
  - Structure `rte_malloc_heap` now has a new field indicating socket ID the malloc heap belongs to
  - Structure `rte_mem_config` has had its `malloc_heaps` array resized from `RTE_MAX_NUMA_NODES` to `RTE_MAX_HEAPS` value
  - Structure `rte_malloc_heap` now has a `heap_name` member
  - Structure `rte_eal_memconfig` has been extended to contain next socket ID for externally allocated segments
- eal: Added `dma_maskbits` to `rte_mem_config` for keeping the most restrictive DMA mask based on the devices addressing limitations.
- eal: The structure `rte_device` has a new field to reference a `rte_bus`. It thus changes the size of the `struct rte_device` and the inherited device structures of all buses.

19.7.4 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_bbdev.so.1
  librte_bitratestats.so.2
  librte_bpf.so.1
+ librte_bus_dpaa.so.2
+ librte_bus_fslmc.so.2
+ librte_bus_ifpga.so.2
+ librte_bus_pci.so.2
+ librte_bus_vdev.so.2
+ librte_bus_vmbus.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_compressdev.so.1
  librte_cryptodev.so.5
  librte_distributor.so.1
+ librte_eal.so.9
  librte_efd.so.1
+ librte_ethdev.so.11
+ librte_eventdev.so.6
  librte_flow_classify.so.1
  librte_gro.so.1
  librte_gso.so.1
  librte_hash.so.2
```

(continues on next page)

(continued from previous page)

```
librte_ip_frag.so.1
librte_jobstats.so.1
librte_kni.so.2
librte_kvargs.so.1
librte_latencystats.so.1
librte_lpm.so.2
librte_mbuf.so.4
librte_member.so.1
librte_mempool.so.5
librte_meter.so.2
librte_metrics.so.1
librte_net.so.1
librte_pci.so.1
librte_pdump.so.2
librte_pipeline.so.3
librte_pmd_bnxt.so.2
librte_pmd_bond.so.2
librte_pmd_i40e.so.2
librte_pmd_ixgbe.so.2
librte_pmd_dpaa2_qdma.so.1
librte_pmd_ring.so.2
librte_pmd_softnic.so.1
librte_pmd_vhost.so.2
librte_port.so.3
librte_power.so.1
librte_rawdev.so.1
librte_reorder.so.1
librte_ring.so.2
librte_sched.so.1
librte_security.so.1
librte_table.so.3
librte_timer.so.1
librte_vhost.so.4
```

19.7.5 Known Issues

- When using SR-IOV (VF) support with netvsc PMD and the Mellanox mlx5 bifurcated driver the Linux netvsc device must be brought up before the netvsc device is unbound and passed to the DPDK.
- IBM Power8 is not supported in this release of DPDK. IBM Power9 is supported.
- `AVX-512` support has been disabled for `GCC` builds [1] because of a crash [2]. This can affect `native` machine type build targets on the platforms that support `AVX512F` like `Intel Skylake` processors, and can cause a possible performance drop. The immediate workaround is to use `clang` compiler on these platforms. The issue has been identified as a GCC defect and reported to the GCC community [3]. Further actions will be taken based on the GCC defect result.
  - [1]: Commit 8d07c82b239f ("mk: disable gcc AVX512F support")
  - [2]: https://bugs.dpdk.org/show_bug.cgi?id=97
  - [3]: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=88096

19.7.6 Tested Platforms

- Intel(R) platforms with Intel(R) NICs combinations
  - CPU
    - Intel(R) Atom(TM) CPU C3758 @ 2.20GHz
    - Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
    - Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
    - Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz
  - OS:
    - CentOS 7.5
    - Fedora 25
    - Fedora 28
    - FreeBSD 11.2
    - Red Hat Enterprise Linux Server release 7.5
    - Open SUSE 15
    - Wind River Linux 8
    - Ubuntu 14.04
    - Ubuntu 16.04
    - Ubuntu 16.10
    - Ubuntu 17.10
    - Ubuntu 18.04
  - NICs:
    - Intel(R) 82599ES 10 Gigabit Ethernet Controller
      - Firmware version: 0x61bf0001
      - Device id (pf/vf): 8086:10fb / 8086:10ed
      - Driver version: 5.2.3 (ixgbe)
    - Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      - Firmware version: 0x800003e7
      - Device id (pf/vf): 8086:15ad / 8086:15a8
      - Driver version: 4.4.6 (ixgbe)
    - Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
      - Firmware version: 6.01 0x80003221
      - Device id (pf/vf): 8086:1572 / 8086:154c

            - Driver version: 2.4.6 (i40e)
        - Intel(R) Corporation Ethernet Connection X722 for 10GbE SFP+ (4x10G)
            - Firmware version: 3.33 0x80000fd5 0.0.0
            - Device id (pf/vf): 8086:37d0 / 8086:37cd
            - Driver version: 2.4.6 (i40e)
        - Intel(R) Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
            - Firmware version: 6.01 0x80003221
            - Device id (pf/vf): 8086:158b / 8086:154c
            - Driver version: 2.4.6 (i40e)
        - Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
            - Firmware version: 6.01 0x8000321c
            - Device id (pf/vf): 8086:1583 / 8086:154c
            - Driver version: 2.4.6 (i40e)
        - Intel(R) Corporation I350 Gigabit Network Connection
            - Firmware version: 1.63, 0x80000dda
            - Device id (pf/vf): 8086:1521 / 8086:1520
            - Driver version: 5.4.0-k (igb)
- Intel(R) platforms with Mellanox(R) NICs combinations
    - CPU:
        - Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz
        - Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
        - Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
        - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
        - Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
        - Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
        - Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
    - OS:
        - Red Hat Enterprise Linux Server release 7.6 (Maipo)
        - Red Hat Enterprise Linux Server release 7.5 (Maipo)
        - Red Hat Enterprise Linux Server release 7.4 (Maipo)
        - Red Hat Enterprise Linux Server release 7.3 (Maipo)
        - Red Hat Enterprise Linux Server release 7.2 (Maipo)
        - Ubuntu 18.10
        - Ubuntu 18.04

    - Ubuntu 17.10
    - Ubuntu 16.04
    - SUSE Linux Enterprise Server 15
- MLNX_OFED: 4.4-2.0.1.0
- MLNX_OFED: 4.5-0.3.1.0
- NICs:
    - Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1007
        - Firmware version: 2.42.5000
    - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.23.8022 and above
    - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.23.8022 and above
    - Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.23.8022 and above
    - Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.23.8022 and above
    - Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.23.8022 and above
    - Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1013
        - Firmware version: 12.23.8022 and above

* Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.23.8022 and above
* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.23.8022 and above
* Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.23.8022 and above
  - Firmware version: 12.23.8022 and above
* Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.23.8022 and above
* Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.23.8022 and above
* Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.23.8022 and above
* Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.23.8022 and above
* Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1017
  - Firmware version: 16.23.8022 and above
* Mellanox(R) ConnectX(R)-5 Ex EN 100G MCX516A-CDAT (2x100G)

            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1019
            - Firmware version: 16.23.8022 and above
- ARM platforms with Mellanox(R) NICs combinations
  - CPU:
    - Qualcomm ARM 1.1 2500MHz
  - OS:
    - Red Hat Enterprise Linux Server release 7.5 (Maipo)
  - NICs:
    - Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1015
      - Firmware version: 14.24.0220
    - Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
      - Host interface: PCI Express 3.0 x16
      - Device ID: 15b3:1017
      - Firmware version: 16.24.0220
- Mellanox(R) BlueField SmartNIC
  - Mellanox(R) BlueField SmartNIC MT416842 (2x25G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:a2d2
    - Firmware version: 18.24.0246
  - SoC ARM cores running OS:
    - CentOS Linux release 7.4.1708 (AltArch)
    - MLNX_OFED 4.4-2.5.3.0
  - DPDK application running on ARM cores inside SmartNIC
- ARM SoC combinations from NXP (with integrated NICs)
  - SoC:
    - NXP/Freescale QorIQ LS1046A with ARM Cortex A72
    - NXP/Freescale QorIQ LS2088A with ARM Cortex A72
  - OS:
    - Ubuntu 18.04.1 LTS with NXP QorIQ LSDK 1809 support packages
    - Ubuntu 16.04.3 LTS with NXP QorIQ LSDK 1803 support packages

19.8 DPDK Release 18.08

19.8.1 New Features

- **Added support for Hyper-V netvsc PMD.**

  The new `netvsc` poll mode driver provides native support for networking on Hyper-V. See the *Netvsc poll mode driver* NIC driver guide for more details on this new driver.

- **Added Flow API support for CXGBE PMD.**

  Flow API support has been added to CXGBE Poll Mode Driver to offload flows to Chelsio T5/T6 NICs. Support added for:

  - Wildcard (LE-TCAM) and Exact (HASH) match filters.
  - Match items: physical ingress port, IPv4, IPv6, TCP and UDP.
  - Action items: queue, drop, count, and physical egress port redirect.

- **Added ixgbe preferred Rx/Tx parameters.**

  Rather than applications providing explicit Rx and Tx parameters such as queue and burst sizes, they can request that the EAL instead uses preferred values provided by the PMD, falling back to defaults within the EAL if the PMD does not provide any. The provision of such tuned values now includes the ixgbe PMD.

- **Added descriptor status check support for fm10k.**

  The `rte_eth_rx_descriptor_status` and `rte_eth_tx_descriptor_status` APIs are now supported by fm10K.

- **Updated the enic driver.**

  - Add low cycle count Tx handler for no-offload Tx.
  - Add low cycle count Rx handler for non-scattered Rx.
  - Minor performance improvements to scattered Rx handler.
  - Add handlers to add/delete VxLAN port number.
  - Add devarg to specify ingress VLAN rewrite mode.

- **Updated mlx5 driver.**

  Updated the mlx5 driver including the following changes:

  - Added port representors support.
  - Added Flow API support for e-switch rules. Added support for ACTION_PORT_ID, ACTION_DROP, ACTION_OF_POP_VLAN, ACTION_OF_PUSH_VLAN, ACTION_OF_SET_VLAN_VID, ACTION_OF_SET_VLAN_PCP and ITEM_PORT_ID.
  - Added support for 32-bit compilation.

- **Added TSO support for the mlx4 driver.**

  Added TSO support for the mlx4 drivers from MLNX_OFED_4.4 and above.

- **SoftNIC PMD rework.**

The SoftNIC PMD infrastructure has been restructured to use the Packet Framework, which makes it more flexible, modular and easier to add new functionality in the future.

- **Updated the AESNI MB PMD.**

  The AESNI MB PMD has been updated with additional support for:

  - 3DES for 8, 16 and 24 byte keys.

- **Added a new compression PMD using Intel's QuickAssist (QAT) device family.**

  Added the new `QAT` compression driver, for compression and decompression operations in software. See the *Intel(R) QuickAssist (QAT) Compression Poll Mode Driver* compression driver guide for details on this new driver.

- **Updated the ISA-L PMD.**

  Added support for chained mbufs (input and output).

19.8.2 API Changes

- The path to the runtime config file has changed. The new path is determined as follows:
  - If DPDK is running as root, `/var/run/dpdk/<prefix>/config`
  - If DPDK is not running as root:
    - If `$XDG_RUNTIME_DIR` is set, `${XDG_RUNTIME_DIR}/dpdk/<prefix>/config`
    - Otherwise, `/tmp/dpdk/<prefix>/config`
- eal: The function `rte_eal_mbuf_default_mempool_ops` was deprecated and is removed in 18.08. It shall be replaced by `rte_mbuf_best_mempool_ops`.
- mempool: Following functions were deprecated and are removed in 18.08:
  - `rte_mempool_populate_iova_tab`
  - `rte_mempool_populate_phys_tab`
  - `rte_mempool_populate_phys` (`rte_mempool_populate_iova` should be used)
  - `rte_mempool_virt2phy` (`rte_mempool_virt2iova` should be used)
  - `rte_mempool_xmem_create`
  - `rte_mempool_xmem_size`
  - `rte_mempool_xmem_usage`
- ethdev: The old offload API is removed:
  - Rx per-port `rte_eth_conf.rxmode.[bit-fields]`
  - Tx per-queue `rte_eth_txconf.txq_flags`
  - `ETH_TXQ_FLAGS_NO*`

  The transition bits are removed:
  - `rte_eth_conf.rxmode.ignore_offload_bitfield`
  - `ETH_TXQ_FLAGS_IGNORE`
- cryptodev: The following API changes have been made in 18.08:

  - In struct `struct rte_cryptodev_info`, field `rte_pci_device *pci_dev` has been replaced with field `struct rte_device *device`.
  - Value 0 is accepted in `sym.max_nb_sessions`, meaning that a device supports an unlimited number of sessions.
  - Two new fields of type `uint16_t` have been added: `min_mbuf_headroom_req` and `min_mbuf_tailroom_req`. These parameters specify the recommended headroom and tailroom for mbufs to be processed by the PMD.

- cryptodev: The following functions were deprecated and are removed in 18.08:
  - `rte_cryptodev_queue_pair_start`
  - `rte_cryptodev_queue_pair_stop`
  - `rte_cryptodev_queue_pair_attach_sym_session`
  - `rte_cryptodev_queue_pair_detach_sym_session`
- cryptodev: The following functions were deprecated and are replaced by other functions in 18.08:
  - `rte_cryptodev_get_header_session_size` is replaced with `rte_cryptodev_sym_get_header_session_size`
  - `rte_cryptodev_get_private_session_size` is replaced with `rte_cryptodev_sym_get_private_session_size`
- cryptodev: Feature flag `RTE_CRYPTODEV_FF_MBUF_SCATTER_GATHER` is replaced with the following more explicit flags:
  - `RTE_CRYPTODEV_FF_IN_PLACE_SGL`
  - `RTE_CRYPTODEV_FF_OOP_SGL_IN_SGL_OUT`
  - `RTE_CRYPTODEV_FF_OOP_SGL_IN_LB_OUT`
  - `RTE_CRYPTODEV_FF_OOP_LB_IN_SGL_OUT`
  - `RTE_CRYPTODEV_FF_OOP_LB_IN_LB_OUT`
- cryptodev: Renamed cryptodev experimental APIs:

  Used user_data instead of private_data in following APIs to avoid confusion with the existing session parameter `sess_private_data[]` and related APIs.
  - `rte_cryptodev_sym_session_set_private_data()` changed to `rte_cryptodev_sym_session_set_user_data()`
  - `rte_cryptodev_sym_session_get_private_data()` changed to `rte_cryptodev_sym_session_get_user_data()`
- compressdev: Feature flag `RTE_COMP_FF_MBUF_SCATTER_GATHER` is replaced with the following more explicit flags:
  - `RTE_COMP_FF_OOP_SGL_IN_SGL_OUT`
  - `RTE_COMP_FF_OOP_SGL_IN_LB_OUT`
  - `RTE_COMP_FF_OOP_LB_IN_SGL_OUT`

19.8.3 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_bbdev.so.1
  librte_bitratestats.so.2
  librte_bpf.so.1
  librte_bus_dpaa.so.1
  librte_bus_fslmc.so.1
  librte_bus_pci.so.1
  librte_bus_vdev.so.1
+ librte_bus_vmbus.so.1
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_common_octeontx.so.1
  librte_compressdev.so.1
+ librte_cryptodev.so.5
  librte_distributor.so.1
+ librte_eal.so.8
+ librte_ethdev.so.10
+ librte_eventdev.so.5
  librte_flow_classify.so.1
  librte_gro.so.1
  librte_gso.so.1
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_latencystats.so.1
  librte_lpm.so.2
  librte_mbuf.so.4
+ librte_mempool.so.5
  librte_meter.so.2
  librte_metrics.so.1
  librte_net.so.1
  librte_pci.so.1
  librte_pdump.so.2
  librte_pipeline.so.3
  librte_pmd_bnxt.so.2
  librte_pmd_bond.so.2
  librte_pmd_i40e.so.2
  librte_pmd_ixgbe.so.2
  librte_pmd_dpaa2_cmdif.so.1
  librte_pmd_dpaa2_qdma.so.1
  librte_pmd_ring.so.2
  librte_pmd_softnic.so.1
  librte_pmd_vhost.so.2
  librte_port.so.3
  librte_power.so.1
  librte_rawdev.so.1
  librte_reorder.so.1
  librte_ring.so.2
  librte_sched.so.1
  librte_security.so.1
  librte_table.so.3
  librte_timer.so.1
  librte_vhost.so.3
```

19.8.4 Tested Platforms

- Intel(R) platforms with Intel(R) NICs combinations
  - CPU
    - Intel(R) Atom(TM) CPU C3858 @ 2.00GHz
    - Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
    - Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-2658 v2 @ 2.40GHz
    - Intel(R) Xeon(R) CPU E5-2658 v3 @ 2.20GHz
    - Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz
  - OS:
    - CentOS 7.4
    - Fedora 25
    - Fedora 27
    - Fedora 28
    - FreeBSD 11.1
    - Red Hat Enterprise Linux Server release 7.5
    - SUSE Enterprise Linux 12
    - Wind River Linux 8
    - Ubuntu 14.04
    - Ubuntu 16.04
    - Ubuntu 16.10
    - Ubuntu 17.10
    - Ubuntu 18.04
  - NICs:
    - Intel(R) 82599ES 10 Gigabit Ethernet Controller
      - Firmware version: 0x61bf0001
      - Device id (pf/vf): 8086:10fb / 8086:10ed
      - Driver version: 5.2.3 (ixgbe)
    - Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      - Firmware version: 0x800003e7
      - Device id (pf/vf): 8086:15ad / 8086:15a8

            - Driver version: 4.4.6 (ixgbe)
        - Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
            - Firmware version: 6.01 0x80003221
            - Device id (pf/vf): 8086:1572 / 8086:154c
            - Driver version: 2.4.6 (i40e)
        - Intel Corporation Ethernet Connection X722 for 10GbE SFP+ (4x10G)
            - Firmware version: 3.33 0x80000fd5 0.0.0
            - Device id (pf/vf): 8086:37d0 / 8086:37cd
            - Driver version: 2.4.3 (i40e)
        - Intel(R) Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
            - Firmware version: 6.01 0x80003221
            - Device id (pf/vf): 8086:158b / 8086:154c
            - Driver version: 2.4.6 (i40e)
        - Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
            - Firmware version: 6.01 0x8000321c
            - Device id (pf/vf): 8086:1583 / 8086:154c
            - Driver version: 2.4.6 (i40e)
        - Intel(R) Corporation I350 Gigabit Network Connection
            - Firmware version: 1.63, 0x80000dda
            - Device id (pf/vf): 8086:1521 / 8086:1520
            - Driver version: 5.4.0-k (igb)
- Intel(R) platforms with Mellanox(R) NICs combinations
    - CPU:
        - Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz
        - Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
        - Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
        - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
        - Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
        - Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
        - Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
    - OS:
        - Red Hat Enterprise Linux Server release 7.5 (Maipo)
        - Red Hat Enterprise Linux Server release 7.4 (Maipo)
        - Red Hat Enterprise Linux Server release 7.3 (Maipo)

    - Red Hat Enterprise Linux Server release 7.2 (Maipo)
    - Ubuntu 18.04
    - Ubuntu 17.10
    - Ubuntu 16.04
    - SUSE Linux Enterprise Server 15
  - MLNX_OFED: 4.3-2.0.2.0
  - MLNX_OFED: 4.4-2.0.1.0
  - NICs:
    - Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1007
      - Firmware version: 2.42.5000
    - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
      - Host interface: PCI Express 3.0 x16

- Device ID: 15b3:1013
- Firmware version: 12.21.1000 and above

* Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.21.1000 and above
* Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.21.1000 and above
* Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1017

            - Firmware version: 16.21.1000 and above
        - Mellanox(R) ConnectX-5 Ex EN 100G MCX516A-CDAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1019
            - Firmware version: 16.21.1000 and above
- ARM platforms with Mellanox(R) NICs combinations
    - CPU:
        - Qualcomm ARM 1.1 2500MHz
    - OS:
        - Red Hat Enterprise Linux Server release 7.5 (Maipo)
    - NICs:
        - Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.23.1000
        - Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.23.1000
- Mellanox BlueField SmartNIC
    - **Mellanox(R) BlueField SmartNIC MT416842 (2x25G)**
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:a2d2
        - Firmware version: 18.99.3950
    - SoC ARM cores running OS: * CentOS Linux release 7.4.1708 (AltArch) * Mellanox MLNX_OFED 4.2-1.4.21.0
    - DPDK application running on ARM cores inside SmartNIC
    - BlueField representors support planned for next release.

19.9 DPDK Release 18.05

19.9.1 New Features

- **Reworked memory subsystem.**

  Memory subsystem has been reworked to support new functionality.

  On Linux, support for reserving/unreserving hugepage memory at runtime has been added, so applications no longer need to pre-reserve memory at startup. Due to reorganized internal workings of memory subsystem, any memory allocated through `rte_malloc()` or `rte_memzone_reserve()` is no longer guaranteed to be IOVA-contiguous.

  This functionality has introduced the following changes:

  - `rte_eal_get_physmem_layout()` was removed.
  - A new flag for memzone reservation (`RTE_MEMZONE_IOVA_CONTIG`) was added to ensure reserved memory will be IOVA-contiguous, for use with device drivers and other cases requiring such memory.
  - New callbacks for memory allocation/deallocation events, allowing users (or drivers) to be notified of new memory being allocated or deallocated
  - New callbacks for validating memory allocations above a specified limit, allowing user to permit or deny memory allocations.
  - A new command-line switch `--legacy-mem` to enable EAL behavior similar to how older versions of DPDK worked (memory segments that are IOVA-contiguous, but hugepages are reserved at startup only, and can never be released).
  - A new command-line switch `--single-file-segments` to put all memory segments within a segment list in a single file.
  - A set of convenience function calls to look up and iterate over allocated memory segments.
  - `-m` and `--socket-mem` command-line arguments now carry an additional meaning and mark pre-reserved hugepages as "unfree-able", thereby acting as a mechanism guaranteeing minimum availability of hugepage memory to the application.

  Reserving/unreserving memory at runtime is not currently supported on FreeBSD.

- **Added bucket mempool driver.**

  Added a bucket mempool driver which provides a way to allocate contiguous block of objects. The number of objects in the block depends on how many objects fit in the `RTE_DRIVER_MEMPOOL_BUCKET_SIZE_KB` memory chunk which is a build time option. The number may be obtained using `rte_mempool_ops_get_info()` API. Contiguous blocks may be allocated using `rte_mempool_get_contig_blocks()` API.

- **Added support for port representors.**

  Added DPDK port representors (also known as "VF representors" in the specific context of VFs), which are to DPDK what the Ethernet switch device driver model (**switchdev**) is to Linux, and which can be thought as a software "patch panel" front-end for applications. DPDK port representors are implemented as additional virtual Ethernet device (**ethdev**) instances, spawned on an as-needed basis through configuration parameters passed to the driver of the underlying device using devargs.

- **Added support for VXLAN and NVGRE tunnel endpoint.**

  New actions types have been added to support encapsulation and decapsulation operations for a tunnel endpoint. The new action types are `RTE_FLOW_ACTION_TYPE_[VXLAN/NVGRE]_ENCAP`, `RTE_FLOW_ACTION_TYPE_[VXLAN/NVGRE]_DECAP`, `RTE_FLOW_ACTION_TYPE_JUMP`. A new item type `RTE_FLOW_ACTION_TYPE_MARK` has been added to match a flow against a previously marked flow. A shared counter has also been introduced to the flow API to count a group of flows.

- **Added PMD-recommended Tx and Rx parameters.**

  Applications can now query drivers for device-tuned values of ring sizes, burst sizes, and number of queues.

- **Added RSS hash and key update to CXGBE PMD.**

  Added support for updating the RSS hash and key to the CXGBE PMD.

- **Added CXGBE VF PMD.**

  CXGBE VF Poll Mode Driver has been added to run DPDK over Chelsio T5/T6 NIC VF instances.

- **Updated mlx5 driver.**

  Updated the mlx5 driver including the following changes:

  - Introduced Multi-packet Rx to enable 100Gb/sec with 64B frames.
  - Support for being run by non-root users given a reduced set of capabilities `CAP_NET_ADMIN`, `CAP_NET_RAW` and `CAP_IPC_LOCK`.
  - Support for TSO and checksum for generic UDP and IP tunnels.
  - Support for inner checksum and RSS for GRE, VXLAN-GPE, MPLSoGRE and MPLSoUDP tunnels.
  - Accommodate the new memory hotplug model.
  - Support for non virtually contiguous mempools.
  - Support for MAC adding along with allmulti and promiscuous modes from VF.
  - Support for Mellanox BlueField SoC device.
  - Support for PMD defaults for queue number and depth to improve the out of the box performance.

- **Updated mlx4 driver.**

  Updated the mlx4 driver including the following changes:

  - Support for to being run by non-root users given a reduced set of capabilities `CAP_NET_ADMIN`, `CAP_NET_RAW` and `CAP_IPC_LOCK`.
  - Supported CRC strip toggling.
  - Accommodate the new memory hotplug model.
  - Support non virtually contiguous mempools.
  - Dropped support for Mellanox OFED 4.2.

- **Updated Solarflare network PMD.**

  Updated the sfc_efx driver including the following changes:

  - Added support for Solarflare XtremeScale X2xxx family adapters.
  - Added support for NVGRE, VXLAN and GENEVE filters in flow API.
  - Added support for DROP action in flow API.
  - Added support for equal stride super-buffer Rx mode (X2xxx only).
  - Added support for MARK and FLAG actions in flow API (X2xxx only).

- **Added Ethernet poll mode driver for AMD XGBE devices.**

  Added the new `axgbe` ethernet poll mode driver for AMD XGBE devices. See the *AXGBE Poll Mode Driver* nic driver guide for more details on this new driver.

- **Updated szedata2 PMD.**

  Added support for new NFB-200G2QL card. A new API was introduced in the libsze2 library which the szedata2 PMD depends on, thus the new version of the library was needed. New versions of the packages are available and the minimum required version is 4.4.1.

- **Added support for Broadcom NetXtreme-S (BCM58800) family of controllers (aka Stingray).**

  Added support for the Broadcom NetXtreme-S (BCM58800) family of controllers (aka Stingray). The BCM58800 devices feature a NetXtreme E-Series advanced network controller, a high-performance ARM CPU block, PCI Express (PCIe) Gen3 interfaces, key accelerators for compute offload and a high-speed memory subsystem including L3 cache and DDR4 interfaces, all interconnected by a coherent Network-on-chip (NOC) fabric.

  The ARM CPU subsystem features eight ARMv8 Cortex-A72 CPUs at 3.0 GHz, arranged in a multi-cluster configuration.

- **Added vDPA in vhost-user lib.**

  Added support for selective datapath in the vhost-user lib. vDPA stands for vhost Data Path Acceleration. It supports virtio ring compatible devices to serve the virtio driver directly to enable datapath acceleration.

- **Added IFCVF vDPA driver.**

  Added IFCVF vDPA driver to support Intel FPGA 100G VF devices. IFCVF works as a HW vhost data path accelerator, it supports live migration and is compatible with virtio 0.95 and 1.0. This driver registers the ifcvf vDPA driver to vhost lib, when virtio connects. With the help of the registered vDPA driver the assigned VF gets configured to Rx/Tx directly to VM's virtio vrings.

- **Added support for vhost dequeue interrupt mode.**

  Added support for vhost dequeue interrupt mode to release CPUs to others when there is no data to transmit. Applications can register an epoll event file descriptor to associate Rx queues with interrupt vectors.

- **Added support for virtio-user server mode.**

  In a container environment if the vhost-user backend restarts, there's no way for it to reconnect to virtio-user. To address this, support for server mode has been added. In this mode the socket file is created by virtio-user, which the backend connects to. This means that if the backend restarts, it can reconnect to virtio-user and continue communications.

- **Added crypto workload support to vhost library.**

New APIs have been introduced in the vhost library to enable virtio crypto support including session creation/deletion handling and translating virtio-crypto requests into DPDK crypto operations. A sample application has also been introduced.

- **Added virtio crypto PMD.**

  Added a new Poll Mode Driver for virtio crypto devices, which provides AES-CBC ciphering and AES-CBC with HMAC-SHA1 algorithm-chaining. See the *Virtio Crypto Poll Mode Driver* crypto driver guide for more details on this new driver.

- **Added AMD CCP Crypto PMD.**

  Added the new `ccp` crypto driver for AMD CCP devices. See the *AMD CCP Poll Mode Driver* crypto driver guide for more details on this new driver.

- **Updated AESNI MB PMD.**

  The AESNI MB PMD has been updated with additional support for:

  - AES-CMAC (128-bit key).

- **Added the Compressdev Library, a generic compression service library.**

  Added the Compressdev library which provides an API for offload of compression and decompression operations to hardware or software accelerator devices.

- **Added a new compression poll mode driver using Intels ISA-L.**

  Added the new `ISA-L` compression driver, for compression and decompression operations in software. See the *ISA-L Compression Poll Mode Driver* compression driver guide for details on this new driver.

- **Added the Event Timer Adapter Library.**

  The Event Timer Adapter Library extends the event-based model by introducing APIs that allow applications to arm/cancel event timers that generate timer expiry events. This new type of event is scheduled by an event device along with existing types of events.

- **Added OcteonTx TIM Driver (Event timer adapter).**

  The OcteonTx Timer block enables software to schedule events for a future time, it is exposed to an application via the Event timer adapter library.

  See the *OCTEON TX SSOVF Eventdev Driver* guide for more details

- **Added Event Crypto Adapter Library.**

  Added the Event Crypto Adapter Library. This library extends the event-based model by introducing APIs that allow applications to enqueue/dequeue crypto operations to/from cryptodev as events scheduled by an event device.

- **Added Ifpga Bus, a generic Intel FPGA Bus library.**

  Added the Ifpga Bus library which provides support for integrating any Intel FPGA device with the DPDK framework. It provides Intel FPGA Partial Bit Stream AFU (Accelerated Function Unit) scan and drivers probe.

- **Added IFPGA (Intel FPGA) Rawdev Driver.**

  Added a new Rawdev driver called IFPGA (Intel FPGA) Rawdev Driver, which cooperates with OPAE (Open Programmable Acceleration Engine) shared code to provide common FPGA management ops for FPGA operation.

See the *IFPGA Rawdev Driver* programmer's guide for more details.

- **Added DPAA2 QDMA Driver (in rawdev).**

  The DPAA2 QDMA is an implementation of the rawdev API, that provide a means of initiating a DMA transaction from CPU. The initiated DMA is performed without the CPU being involved in the actual DMA transaction.

  See the *NXP DPAA2 QDMA Driver* guide for more details.

- **Added DPAA2 Command Interface Driver (in rawdev).**

  The DPAA2 CMDIF is an implementation of the rawdev API, that provides communication between the GPP and NXP's QorIQ based AIOP Block (Firmware). Advanced IO Processor i.e. AIOP are clusters of programmable RISC engines optimized for flexible networking and I/O operations. The communication between GPP and AIOP is achieved via using DPCI devices exposed by MC for GPP <–> AIOP interaction.

  See the *NXP DPAA2 CMDIF Driver* guide for more details.

- **Added device event monitor framework.**

  Added a general device event monitor framework to EAL, for device dynamic management to facilitate device hotplug awareness and associated actions. The list of new APIs is:

  - `rte_dev_event_monitor_start` and `rte_dev_event_monitor_stop` for the event monitor enabling and disabling.
  - `rte_dev_event_callback_register` and `rte_dev_event_callback_unregister` for registering and un-registering user callbacks.

  Linux uevent is supported as a backend of this device event notification framework.

- **Added support for procinfo and pdump on eth vdev.**

  For ethernet virtual devices (like TAP, PCAP, etc.), with this feature, we can get stats/xstats on shared memory from a secondary process, and also pdump packets on those virtual devices.

- **Enhancements to the Packet Framework Library.**

  Design and development of new API functions for Packet Framework library that implement a common set of actions such as traffic metering, packet encapsulation, network address translation, TTL update, etc., for pipeline table and input ports to speed up application development. The API functions includes creating action profiles, registering actions to the profiles, instantiating action profiles for pipeline table and input ports, etc.

- **Added the BPF Library.**

  The BPF Library provides the ability to load and execute Enhanced Berkeley Packet Filters (eBPF) within user-space DPDK applications. It also introduces a basic framework to load/unload BPF-based filters on Eth devices (right now only via SW RX/TX callbacks). It also adds a dependency on libelf.

19.9.2 API Changes

- service cores: No longer marked as experimental.

  The service cores functions are no longer marked as experimental, and have become part of the normal DPDK API and ABI. Any future ABI changes will be announced at least one release before the ABI change is made. There are no ABI breaking changes planned.

- eal: The `rte_lcore_has_role()` return value changed.

  This function now returns true or false, respectively, rather than 0 or < 0 for success or failure. It makes use of the function more intuitive.

- mempool: The capability flags and related functions have been removed.

  Flags `MEMPOOL_F_CAPA_PHYS_CONTIG` and `MEMPOOL_F_CAPA_BLK_ALIGNED_OBJECTS` were used by octeontx mempool driver to customize generic mempool library behavior. Now the new driver callbacks `calc_mem_size` and `populate` may be used to achieve it without specific knowledge in the generic code.

- mempool: The following xmem functions have been deprecated:

  - `rte_mempool_xmem_create`
  - `rte_mempool_xmem_size`
  - `rte_mempool_xmem_usage`
  - `rte_mempool_populate_iova_tab`

- mbuf: The control mbuf API has been removed in v18.05. The impacted functions and macros are:

  - `rte_ctrlmbuf_init()`
  - `rte_ctrlmbuf_alloc()`
  - `rte_ctrlmbuf_free()`
  - `rte_ctrlmbuf_data()`
  - `rte_ctrlmbuf_len()`
  - `rte_is_ctrlmbuf()`
  - `CTRL_MBUF_FLAG`

  The packet mbuf API should be used as a replacement.

- meter: API updated to accommodate configuration profiles.

  The meter API has been changed to support meter configuration profiles. The configuration profile represents the set of configuration parameters for a given meter object, such as the rates and sizes for the token buckets. These configuration parameters were previously part of the meter object internal data structure. The separation of the configuration parameters from the meter object data structure results in reducing its memory footprint which helps in better cache utilization when a large number of meter objects are used.

- ethdev: The function `rte_eth_dev_count()`, often mis-used to iterate over ports, is deprecated and replaced by `rte_eth_dev_count_avail()`. There is also a new function `rte_eth_dev_count_total()` to get the total number of allocated ports, avail-

able or not. The hotplug-proof applications should use RTE_ETH_FOREACH_DEV or RTE_ETH_FOREACH_DEV_OWNED_BY as port iterators.

- ethdev: In struct struct rte_eth_dev_info, field rte_pci_device *pci_dev has been replaced with field struct rte_device *device.
- ethdev: Changes to the semantics of rte_eth_dev_configure() parameters.

  If both the nb_rx_q and nb_tx_q parameters are zero, rte_eth_dev_configure() will now use PMD-recommended queue sizes, or if recommendations are not provided by the PMD the function will use ethdev fall-back values. Previously setting both of the parameters to zero would have resulted in -EINVAL being returned.
- ethdev: Changes to the semantics of rte_eth_rx_queue_setup() parameters.

  If the nb_rx_desc parameter is zero, rte_eth_rx_queue_setup will now use the PMD-recommended Rx ring size, or in the case where the PMD does not provide a recommendation, will use an ethdev-provided fall-back value. Previously, setting nb_rx_desc to zero would have resulted in an error.
- ethdev: Changes to the semantics of rte_eth_tx_queue_setup() parameters.

  If the nb_tx_desc parameter is zero, rte_eth_tx_queue_setup will now use the PMD-recommended Tx ring size, or in the case where the PMD does not provide a recommendation, will use an ethdev-provided fall-back value. Previously, setting nb_tx_desc to zero would have resulted in an error.
- ethdev: Several changes were made to the flow API.
  - The unused DUP action was removed.
  - Actions semantics in flow rules: list order now matters ("first to last" instead of "all simultaneously"), repeated actions are now all performed, and they do not individually have (non-)terminating properties anymore.
  - Flow rules are now always terminating unless a PASSTHRU action is present.
  - C99-style flexible arrays were replaced with standard pointers in RSS action and in RAW pattern item structures due to compatibility issues.
  - The RSS action was modified to not rely on external struct rte_eth_rss_conf anymore to instead expose its own and more appropriately named configuration fields directly (rss_conf->rss_key => key, rss_conf->rss_key_len => key_len, rss_conf->rss_hf => types, num => queue_num), and the addition of missing RSS parameters (func for RSS hash function to apply and level for the encapsulation level).
  - The VLAN pattern item (struct rte_flow_item_vlan) was modified to include inner EtherType instead of outer TPID. Its default mask was also modified to cover the VID part (lower 12 bits) of TCI only.
  - A new transfer attribute was added to struct rte_flow_attr in order to clarify the behavior of some pattern items.
  - PF and VF pattern items are now only accepted by PMDs that implement them (bnxt and i40e) when the transfer attribute is also present, for consistency.
  - Pattern item PORT was renamed PHY_PORT to avoid confusion with DPDK port IDs.
  - An action counterpart to the PHY_PORT pattern item was added in order to redirect matching traffic to a specific physical port.

  - PORT_ID pattern item and actions were added to match and target DPDK port IDs at a higher level than PHY_PORT.
  - `RTE_FLOW_ACTION_TYPE_[VXLAN/NVGRE]_ENCAP` action items were added to support tunnel encapsulation operation for VXLAN and NVGRE type tunnel endpoint.
  - `RTE_FLOW_ACTION_TYPE_[VXLAN/NVGRE]_DECAP` action items were added to support tunnel decapsulation operation for VXLAN and NVGRE type tunnel endpoint.
  - `RTE_FLOW_ACTION_TYPE_JUMP` action item was added to support a matched flow to be redirected to the specific group.
  - `RTE_FLOW_ACTION_TYPE_MARK` item type has been added to match a flow against a previously marked flow.

- ethdev: Change flow APIs regarding count action:
  - `rte_flow_create()` API count action now requires the `struct rte_flow_action_count`.
  - `rte_flow_query()` API parameter changed from action type to action structure.

- ethdev: Changes to offload API

  A pure per-port offloading isn't requested to be repeated in [rt]x_conf->offloads to `rte_eth_[rt]x_queue_setup()`. Now any offloading enabled in `rte_eth_dev_configure()` can't be disabled by `rte_eth_[rt]x_queue_setup()`. Any new added offloading which has not been enabled in `rte_eth_dev_configure()` and is requested to be enabled in `rte_eth_[rt]x_queue_setup()` must be per-queue type, or otherwise trigger an error log.

- ethdev: Runtime queue setup

  `rte_eth_rx_queue_setup` and `rte_eth_tx_queue_setup` can be called after `rte_eth_dev_start` if the device supports runtime queue setup. The device driver can expose this capability through `rte_eth_dev_info_get`. A Rx or Tx queue set up at runtime need to be started explicitly by `rte_eth_dev_rx_queue_start` or `rte_eth_dev_tx_queue_start`.

19.9.3 ABI Changes

- ring: The alignment constraints on the ring structure have been relaxed to one cache line instead of two, and an empty cache line padding is added between the producer and consumer structures. The size of the structure and the offset of the fields remains the same on platforms with 64B cache line, but changes on other platforms.

- mempool: Some ops have changed.

  A new callback `calc_mem_size` has been added to `rte_mempool_ops` to allow customization of the required memory size calculation. A new callback `populate` has been added to `rte_mempool_ops` to allow customized object population. Callback `get_capabilities` has been removed from `rte_mempool_ops` since its features are covered by `calc_mem_size` and `populate` callbacks. Callback `register_memory_area` has been removed from `rte_mempool_ops` since the new callback `populate` may be used instead of it.

- ethdev: Additional fields in rte_eth_dev_info.

The `rte_eth_dev_info` structure has had two extra entries appended to the end of it: `default_rxportconf` and `default_txportconf`. Each of these in turn are `rte_eth_dev_portconf` structures containing three fields of type `uint16_t`: `burst_size`, `ring_size`, and `nb_queues`. These are parameter values recommended for use by the PMD.

- ethdev: ABI for all flow API functions was updated.

  This includes functions `rte_flow_copy`, `rte_flow_create`, `rte_flow_destroy`, `rte_flow_error_set`, `rte_flow_flush`, `rte_flow_isolate`, `rte_flow_query` and `rte_flow_validate`, due to changes in error type definitions (`enum rte_flow_error_type`), removal of the unused DUP action (`enum rte_flow_action_type`), modified behavior for flow rule actions (see API changes), removal of C99 flexible array from RAW pattern item (`struct rte_flow_item_raw`), complete rework of the RSS action definition (`struct rte_flow_action_rss`), sanity fix in the VLAN pattern item (`struct rte_flow_item_vlan`) and new transfer attribute (`struct rte_flow_attr`).

- bbdev: New parameter added to rte_bbdev_op_cap_turbo_dec.

  A new parameter `max_llr_modulus` has been added to `rte_bbdev_op_cap_turbo_dec` structure to specify maximal LLR (likelihood ratio) absolute value.

- bbdev: Queue Groups split into UL/DL Groups.

  Queue Groups have been split into UL/DL Groups in the Turbo Software Driver. They are independent for Decode/Encode. `rte_bbdev_driver_info` reflects introduced changes.

19.9.4 Known Issues

- **Secondary process launch is not reliable.**

  Recent memory hotplug patches have made multiprocess startup less reliable than it was in past releases. A number of workarounds are known to work depending on the circumstances. As such it isn't recommended to use the secondary process mechanism for critical systems. The underlying issues will be addressed in upcoming releases.

  The issue is explained in more detail, including potential workarounds, in the Bugzilla entry referenced below.

  Bugzilla entry: https://bugs.dpdk.org/show_bug.cgi?id=50

- **pdump is not compatible with old applications.**

  As we changed to use generic multi-process communication for pdump negotiation instead of previous dedicated unix socket way, pdump applications, including the dpdk-pdump example and any other applications using `librte_pdump`, will not work with older version DPDK primary applications.

- **rte_abort takes a long time on FreeBSD.**

  DPDK processes now allocates a large area of virtual memory address space. As a result `rte_abort` on FreeBSD now dumps the contents of the whole reserved memory range, not just the used portion, to a core dump file. Writing this large core file can take a significant amount of time, causing processes to appear to hang on the system.

  The work around for the issue is to set the system resource limits for core dumps before running any tests, e.g. `limit coredumpsize 0`. This will effectively disable core dumps on FreeBSD. If they are not to be completely disabled, a suitable limit, e.g. 1G might be specified instead of

0. This needs to be run per-shell session, or before every test run. This change can also be made persistent by adding `kern.coredump=0` to `/etc/sysctl.conf`.

Bugzilla entry: https://bugs.dpdk.org/show_bug.cgi?id=53

- **ixgbe PMD crash on hotplug detach when no VF created.**

  ixgbe PMD uninit path cause null pointer dereference because of port representor cleanup when number of VF is zero.

  Bugzilla entry: https://bugs.dpdk.org/show_bug.cgi?id=57

- **Bonding PMD may fail to accept new slave ports in certain conditions.**

  In certain conditions when using testpmd, bonding may fail to register new slave ports.

  Bugzilla entry: https://bugs.dpdk.org/show_bug.cgi?id=52.

- **Unexpected performance regression in Vhost library.**

  Patches fixing CVE-2018-1059 were expected to introduce a small performance drop. However, in some setups, bigger performance drops have been measured when running micro-benchmarks.

  Bugzilla entry: https://bugs.dpdk.org/show_bug.cgi?id=48

19.9.5 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_bbdev.so.1
  librte_bitratestats.so.2
+ librte_bpf.so.1
  librte_bus_dpaa.so.1
  librte_bus_fslmc.so.1
  librte_bus_pci.so.1
  librte_bus_vdev.so.1
  librte_cfgfile.so.2
  librte_cmdline.so.2
+ librte_common_octeontx.so.1
+ librte_compressdev.so.1
  librte_cryptodev.so.4
  librte_distributor.so.1
+ librte_eal.so.7
+ librte_ethdev.so.9
+ librte_eventdev.so.4
  librte_flow_classify.so.1
  librte_gro.so.1
  librte_gso.so.1
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_latencystats.so.1
  librte_lpm.so.2
+ librte_mbuf.so.4
+ librte_mempool.so.4
+ librte_meter.so.2
  librte_metrics.so.1
  librte_net.so.1
```

(continues on next page)

(continued from previous page)

```
  librte_pci.so.1
  librte_pdump.so.2
  librte_pipeline.so.3
  librte_pmd_bnxt.so.2
  librte_pmd_bond.so.2
  librte_pmd_i40e.so.2
  librte_pmd_ixgbe.so.2
+ librte_pmd_dpaa2_cmdif.so.1
+ librte_pmd_dpaa2_qdma.so.1
  librte_pmd_ring.so.2
  librte_pmd_softnic.so.1
  librte_pmd_vhost.so.2
  librte_port.so.3
  librte_power.so.1
  librte_rawdev.so.1
  librte_reorder.so.1
+ librte_ring.so.2
  librte_sched.so.1
  librte_security.so.1
  librte_table.so.3
  librte_timer.so.1
  librte_vhost.so.3
```

19.9.6 Tested Platforms

- Intel(R) platforms with Intel(R) NICs combinations
  - CPU
    - Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
    - Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
    - Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-2658 v2 @ 2.40GHz
    - Intel(R) Xeon(R) CPU E5-2658 v3 @ 2.20GHz
    - Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz
  - OS:
    - CentOS 7.4
    - Fedora 25
    - Fedora 27
    - Fedora 28
    - FreeBSD 11.1
    - Red Hat Enterprise Linux Server release 7.3
    - SUSE Enterprise Linux 12

    - Wind River Linux 8
    - Ubuntu 14.04
    - Ubuntu 16.04
    - Ubuntu 16.10
    - Ubuntu 17.10
  - NICs:
    - Intel(R) 82599ES 10 Gigabit Ethernet Controller
      - Firmware version: 0x61bf0001
      - Device id (pf/vf): 8086:10fb / 8086:10ed
      - Driver version: 5.2.3 (ixgbe)
    - Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      - Firmware version: 0x800003e7
      - Device id (pf/vf): 8086:15ad / 8086:15a8
      - Driver version: 4.4.6 (ixgbe)
    - Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
      - Firmware version: 6.01 0x80003221
      - Device id (pf/vf): 8086:1572 / 8086:154c
      - Driver version: 2.4.6 (i40e)
    - Intel Corporation Ethernet Connection X722 for 10GbE SFP+ (4x10G)
      - Firmware version: 3.33 0x80000fd5 0.0.0
      - Device id (pf/vf): 8086:37d0 / 8086:37cd
      - Driver version: 2.4.3 (i40e)
    - Intel(R) Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
      - Firmware version: 6.01 0x80003221
      - Device id (pf/vf): 8086:158b / 8086:154c
      - Driver version: 2.4.6 (i40e)
    - Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
      - Firmware version: 6.01 0x8000321c
      - Device id (pf/vf): 8086:1583 / 8086:154c
      - Driver version: 2.4.6 (i40e)
    - Intel(R) Corporation I350 Gigabit Network Connection
      - Firmware version: 1.63, 0x80000dda
      - Device id (pf/vf): 8086:1521 / 8086:1520
      - Driver version: 5.4.0-k (igb)

- Intel(R) platforms with Mellanox(R) NICs combinations
  - CPU:
    - Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz
    - Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
    - Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
    - Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
    - Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
  - OS:
    - Red Hat Enterprise Linux Server release 7.5 (Maipo)
    - Red Hat Enterprise Linux Server release 7.4 (Maipo)
    - Red Hat Enterprise Linux Server release 7.3 (Maipo)
    - Red Hat Enterprise Linux Server release 7.2 (Maipo)
    - Ubuntu 18.04
    - Ubuntu 17.10
    - Ubuntu 16.10
    - Ubuntu 16.04
    - SUSE Linux Enterprise Server 15
  - MLNX_OFED: 4.2-1.0.0.0
  - MLNX_OFED: 4.3-2.0.2.0
  - NICs:
    - Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1007
      - Firmware version: 2.42.5000
    - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.21.1000 and above

* Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
* Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)

        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1013
        - Firmware version: 12.21.1000 and above
      * Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1015
        - Firmware version: 14.21.1000 and above
      * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1015
        - Firmware version: 14.21.1000 and above
      * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1017
        - Firmware version: 16.21.1000 and above
      * Mellanox(R) ConnectX-5 Ex EN 100G MCX516A-CDAT (2x100G)
        - Host interface: PCI Express 4.0 x16
        - Device ID: 15b3:1019
        - Firmware version: 16.21.1000 and above
- ARM platforms with Mellanox(R) NICs combinations
  - CPU:
    * Qualcomm ARM 1.1 2500MHz
  - OS:
    * Red Hat Enterprise Linux Server release 7.5 (Maipo)
  - NICs:
    * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1015
      - Firmware version: 14.22.0428
    * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
      - Host interface: PCI Express 3.0 x16
      - Device ID: 15b3:1017
      - Firmware version: 16.22.0428
- ARM SoC combinations from Cavium (with integrated NICs)

  - SoC:
    * Cavium CN81xx
    * Cavium CN83xx
  - OS:
    * Ubuntu 16.04.2 LTS with Cavium SDK-6.2.0-Patch2 release support package.
- ARM SoC combinations from NXP (with integrated NICs)
  - SoC:
    * NXP/Freescale QorIQ LS1046A with ARM Cortex A72
    * NXP/Freescale QorIQ LS2088A with ARM Cortex A72
  - OS:
    * Ubuntu 16.04.3 LTS with NXP QorIQ LSDK 1803 support packages

19.10 DPDK Release 18.02

19.10.1 New Features

- **Added function to allow releasing internal EAL resources on exit.**

  During `rte_eal_init()` EAL allocates memory from hugepages to enable its core libraries to perform their tasks. The `rte_eal_cleanup()` function releases these resources, ensuring that no hugepage memory is leaked. It is expected that all DPDK applications call `rte_eal_cleanup()` before exiting. Not calling this function could result in leaking hugepages, leading to failure during initialization of secondary processes.

- **Added igb, ixgbe and i40e ethernet driver to support RSS with flow API.**

  Added support for igb, ixgbe and i40e NICs with existing RSS configuration using the `rte_flow` API.

  Also enabled queue region configuration using the `rte_flow` API for i40e.

- **Updated i40e driver to support PPPoE/PPPoL2TP.**

  Updated i40e PMD to support PPPoE/PPPoL2TP with PPPoE/PPPoL2TP supporting profiles which can be programmed by dynamic device personalization (DDP) process.

- **Added MAC loopback support for i40e.**

  Added MAC loopback support for i40e in order to support test tasks requested by users. It will setup `Tx -> Rx` loopback link according to the device configuration.

- **Added support of run time determination of number of queues per i40e VF.**

  The number of queue per VF is determined by its host PF. If the PCI address of an i40e PF is `aaaa:bb.cc`, the number of queues per VF can be configured with EAL parameter like `-w aaaa:bb.cc,queue-num-per-vf=n`. The value n can be 1, 2, 4, 8 or 16. If no such parameter is configured, the number of queues per VF is 4 by default.

- **Updated mlx5 driver.**

  Updated the mlx5 driver including the following changes:

  - Enabled compilation as a plugin, thus removed the mandatory dependency with rdma-core. With the special compilation, the rdma-core libraries will be loaded only in case Mellanox device is being used. For binaries creation the PMD can be enabled, still not requiring from every end user to install rdma-core.
  - Improved multi-segment packet performance.
  - Changed driver name to use the PCI address to be compatible with OVS-DPDK APIs.
  - Extended statistics for physical port packet/byte counters.
  - Converted to the new offloads API.
  - Supported device removal check operation.

- **Updated mlx4 driver.**

  Updated the mlx4 driver including the following changes:

  - Enabled compilation as a plugin, thus removed the mandatory dependency with rdma-core. With the special compilation, the rdma-core libraries will be loaded only in case Mellanox device is being used. For binaries creation the PMD can be enabled, still not requiring from every end user to install rdma-core.
  - Improved data path performance.
  - Converted to the new offloads API.
  - Supported device removal check operation.

- **Added NVGRE and UDP tunnels support in Solarflare network PMD.**

  Added support for NVGRE, VXLAN and GENEVE tunnels.

  - Added support for UDP tunnel ports configuration.
  - Added tunneled packets classification.
  - Added inner checksum offload.

- **Added AVF (Adaptive Virtual Function) net PMD.**

  Added a new net PMD called AVF (Adaptive Virtual Function), which supports Intel® Ethernet Adaptive Virtual Function (AVF) with features such as:

  - Basic Rx/Tx burst
  - SSE vectorized Rx/Tx burst
  - Promiscuous mode
  - MAC/VLAN offload
  - Checksum offload
  - TSO offload
  - Jumbo frame and MTU setting
  - RSS configuration
  - stats
  - Rx/Tx descriptor status
  - Link status update/event

- **Added feature supports for live migration from vhost-net to vhost-user.**

  Added feature supports for vhost-user to make live migration from vhost-net to vhost-user possible. The features include:

  - `VIRTIO_F_ANY_LAYOUT`
  - `VIRTIO_F_EVENT_IDX`
  - `VIRTIO_NET_F_GUEST_ECN`, `VIRTIO_NET_F_HOST_ECN`
  - `VIRTIO_NET_F_GUEST_UFO`, `VIRTIO_NET_F_HOST_UFO`
  - `VIRTIO_NET_F_GSO`

  Also added `VIRTIO_NET_F_GUEST_ANNOUNCE` feature support in virtio pmd. In a scenario where the vhost backend doesn't have the ability to generate RARP packets, the VM running virtio pmd can still be live migrated if `VIRTIO_NET_F_GUEST_ANNOUNCE` feature is negotiated.

- **Updated the AESNI-MB PMD.**

  The AESNI-MB PMD has been updated with additional support for:

  - AES-CCM algorithm.

- **Updated the DPAA_SEC crypto driver to support rte_security.**

  Updated the `dpaa_sec` crypto PMD to support `rte_security` lookaside protocol offload for IPsec.

- **Added Wireless Base Band Device (bbdev) abstraction.**

  The Wireless Baseband Device library is an acceleration abstraction framework for 3gpp Layer 1 processing functions that provides a common programming interface for seamless operation on integrated or discrete hardware accelerators or using optimized software libraries for signal processing.

  The current release only supports 3GPP CRC, Turbo Coding and Rate Matching operations, as specified in 3GPP TS 36.212.

  See the *Wireless Baseband Device Library* programmer's guide for more details.

- **Added New eventdev Ordered Packet Distribution Library (OPDL) PMD.**

  The OPDL (Ordered Packet Distribution Library) eventdev is a specific implementation of the eventdev API. It is particularly suited to packet processing workloads that have high throughput and low latency requirements. All packets follow the same path through the device. The order in which packets follow is determined by the order in which queues are set up. Events are left on the ring until they are transmitted. As a result packets do not go out of order.

  With this change, applications can use the OPDL PMD via the eventdev api.

- **Added new pipeline use case for dpdk-test-eventdev application.**

  Added a new "pipeline" use case for the `dpdk-test-eventdev` application. The pipeline case can be used to simulate various stages in a real world application from packet receive to transmit while maintaining the packet ordering. It can also be used to measure the performance of the event device across the stages of the pipeline.

  The pipeline use case has been made generic to work with all the event devices based on the capabilities.

- **Updated Eventdev sample application to support event devices based on capability.**

  Updated the Eventdev pipeline sample application to support various types of pipelines based on the capabilities of the attached event and ethernet devices. Also, renamed the application from software PMD specific `eventdev_pipeline_sw_pmd` to the more generic `eventdev_pipeline`.

- **Added Rawdev, a generic device support library.**

  The Rawdev library provides support for integrating any generic device type with the DPDK framework. Generic devices are those which do not have a pre-defined type within DPDK, for example, ethernet, crypto, event etc.

  A set of northbound APIs have been defined which encompass a generic set of operations by allowing applications to interact with device using opaque structures/buffers. Also, southbound APIs provide a means of integrating devices either as part of a physical bus (PCI, FSLMC etc) or through `vdev`.

  See the *Rawdevice Library* programmer's guide for more details.

- **Added new multi-process communication channel.**

  Added a generic channel in EAL for multi-process (primary/secondary) communication. Consumers of this channel need to register an action with an action name to response a message received; the actions will be identified by the action name and executed in the context of a new dedicated thread for this channel. The list of new APIs:

  - `rte_mp_register` and `rte_mp_unregister` are for action (un)registration.
  - `rte_mp_sendmsg` is for sending a message without blocking for a response.
  - `rte_mp_request` is for sending a request message and will block until it gets a reply message which is sent from the peer by `rte_mp_reply`.

- **Added GRO support for VxLAN-tunneled packets.**

  Added GRO support for VxLAN-tunneled packets. Supported VxLAN packets must contain an outer IPv4 header and inner TCP/IPv4 headers. VxLAN GRO doesn't check if input packets have correct checksums and doesn't update checksums for output packets. Additionally, it assumes the packets are complete (i.e., `MF==0 && frag_off==0`), when IP fragmentation is possible (i.e., `DF==0`).

- **Increased default Rx and Tx ring size in sample applications.**

  Increased the default `RX_RING_SIZE` and `TX_RING_SIZE` to 1024 entries in testpmd and the sample applications to give better performance in the general case. The user should experiment with various Rx and Tx ring sizes for their specific application to get best performance.

- **Added new DPDK build system using the tools "meson" and "ninja" [EXPERIMENTAL].**

  Added support for building DPDK using `meson` and `ninja`, which gives additional features, such as automatic build-time configuration, over the current build system using `make`. For instructions on how to do a DPDK build using the new system, see the instructions in `doc/build-sdk-meson.txt`.

Note: This new build system support is incomplete at this point and is added as experimental in this release. The existing build system using `make` is unaffected by these changes, and can continue to be used for this and subsequent releases until such time as it's deprecation is announced.

19.10.2 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
+ librte_bbdev.so.1
  librte_bitratestats.so.2
  librte_bus_dpaa.so.1
  librte_bus_fslmc.so.1
  librte_bus_pci.so.1
  librte_bus_vdev.so.1
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_cryptodev.so.4
  librte_distributor.so.1
  librte_eal.so.6
  librte_ethdev.so.8
  librte_eventdev.so.3
  librte_flow_classify.so.1
  librte_gro.so.1
  librte_gso.so.1
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_latencystats.so.1
  librte_lpm.so.2
  librte_mbuf.so.3
  librte_mempool.so.3
  librte_meter.so.1
  librte_metrics.so.1
  librte_net.so.1
  librte_pci.so.1
  librte_pdump.so.2
  librte_pipeline.so.3
  librte_pmd_bnxt.so.2
  librte_pmd_bond.so.2
  librte_pmd_i40e.so.2
  librte_pmd_ixgbe.so.2
  librte_pmd_ring.so.2
  librte_pmd_softnic.so.1
  librte_pmd_vhost.so.2
  librte_port.so.3
  librte_power.so.1
+ librte_rawdev.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
  librte_security.so.1
  librte_table.so.3
  librte_timer.so.1
  librte_vhost.so.3
```

19.10.3 Tested Platforms

- Intel(R) platforms with Intel(R) NICs combinations
  - CPU
    - Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
    - Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
    - Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
    - Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-2658 v2 @ 2.40GHz
    - Intel(R) Xeon(R) CPU E5-2658 v3 @ 2.20GHz
    - Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz
  - OS:
    - CentOS 7.2
    - Fedora 25
    - Fedora 26
    - Fedora 27
    - FreeBSD 11
    - Red Hat Enterprise Linux Server release 7.3
    - SUSE Enterprise Linux 12
    - Wind River Linux 8
    - Ubuntu 14.04
    - Ubuntu 16.04
    - Ubuntu 16.10
    - Ubuntu 17.10
  - NICs:
    - Intel(R) 82599ES 10 Gigabit Ethernet Controller
      - Firmware version: 0x61bf0001
      - Device id (pf/vf): 8086:10fb / 8086:10ed
      - Driver version: 5.2.3 (ixgbe)
    - Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      - Firmware version: 0x800003e7
      - Device id (pf/vf): 8086:15ad / 8086:15a8

            - Driver version: 4.4.6 (ixgbe)
        * Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
            - Firmware version: 6.01 0x80003221
            - Device id (pf/vf): 8086:1572 / 8086:154c
            - Driver version: 2.4.3 (i40e)
        * Intel Corporation Ethernet Connection X722 for 10GBASE-T
            - firmware-version: 6.01 0x80003221
            - Device id: 8086:37d2 / 8086:154c
            - Driver version: 2.4.3 (i40e)
        * Intel(R) Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
            - Firmware version: 6.01 0x80003221
            - Device id (pf/vf): 8086:158b / 8086:154c
            - Driver version: 2.4.3 (i40e)
        * Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
            - Firmware version: 6.01 0x8000321c
            - Device id (pf/vf): 8086:1583 / 8086:154c
            - Driver version: 2.4.3 (i40e)
        * Intel(R) Corporation I350 Gigabit Network Connection
            - Firmware version: 1.63, 0x80000dda
            - Device id (pf/vf): 8086:1521 / 8086:1520
            - Driver version: 5.3.0-k (igb)

- Intel(R) platforms with Mellanox(R) NICs combinations
    - CPU:
        * Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
        * Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
        * Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
        * Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
        * Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
        * Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
    - OS:
        * Red Hat Enterprise Linux Server release 7.5 Beta (Maipo)
        * Red Hat Enterprise Linux Server release 7.4 (Maipo)
        * Red Hat Enterprise Linux Server release 7.3 (Maipo)
        * Red Hat Enterprise Linux Server release 7.2 (Maipo)

    - Ubuntu 17.10
    - Ubuntu 16.10
    - Ubuntu 16.04
- MLNX_OFED: 4.2-1.0.0.0
- MLNX_OFED: 4.3-0.1.6.0
- NICs:
    - Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1007
        - Firmware version: 2.42.5000
    - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.21.1000 and above
    - Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1013
        - Firmware version: 12.21.1000 and above

- Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
- Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
- Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
  - Firmware version: 12.21.1000 and above
- Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
- Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000 and above
- Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.21.1000 and above
- Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.21.1000 and above
- Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1017
  - Firmware version: 16.21.1000 and above
- Mellanox(R) ConnectX-5 Ex EN 100G MCX516A-CDAT (2x100G)

            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1019
            - Firmware version: 16.21.1000 and above

- ARM platforms with Mellanox(R) NICs combinations
  - CPU:
    - Qualcomm ARM 1.1 2500MHz
  - OS:
    - Ubuntu 16.04
  - MLNX_OFED: 4.2-1.0.0.0
  - NICs:
    - Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1015
      - Firmware version: 14.21.1000
    - Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
      - Host interface: PCI Express 3.0 x16
      - Device ID: 15b3:1017
      - Firmware version: 16.21.1000

19.11 DPDK Release 17.11

19.11.1 New Features

- **Extended port_id range from uint8_t to uint16_t.**

  Increased the `port_id` range from 8 bits to 16 bits in order to support more than 256 ports in DPDK. All ethdev APIs which have `port_id` as parameter have been changed.

- **Modified the return type of rte_eth_stats_reset.**

  Changed return type of `rte_eth_stats_reset` from `void` to `int` so that the caller can determine whether a device supports the operation or not and if the operation was carried out.

- **Added a new driver for Marvell Armada 7k/8k devices.**

  Added the new `mrvl` net driver for Marvell Armada 7k/8k devices. See the *MVPP2 Poll Mode Driver* NIC guide for more details on this new driver.

- **Updated mlx4 driver.**

  Updated the mlx4 driver including the following changes:

  - Isolated mode (rte_flow) can now be enabled anytime, not only during initial device configuration.
  - Flow rules now support up to 4096 priority levels usable at will by applications.

  - Enhanced error message to help debugging invalid/unsupported flow rules.
  - Flow rules matching all multicast and promiscuous traffic are now allowed.
  - No more software restrictions on flow rules with the RSS action, their configuration is much more flexible.
  - Significantly reduced memory footprint for Rx and Tx queue objects.
  - While supported, UDP RSS is temporarily disabled due to a remaining issue with its support in the Linux kernel.
  - The new RSS implementation does not automatically spread traffic according to the inner packet of VXLAN frames anymore, only the outer one (like other PMDs).
  - Partial (Tx only) support for secondary processes was broken and had to be removed.
  - Refactored driver to get rid of dependency on the components provided by Mellanox OFED and instead rely on the current and public rdma-core package and Linux version from now on.
  - Removed compile-time limitation on number of device instances the PMD can support.

- **Updated mlx5 driver.**

  Updated the mlx5 driver including the following changes:

  - Enabled the PMD to run on top of upstream Linux kernel and rdma-core libs, removing the dependency on specific Mellanox OFED libraries.
  - Improved PMD latency performance.
  - Improved PMD memory footprint.
  - Added support for vectorized Rx/Tx burst for ARMv8.
  - Added support for secondary process.
  - Added support for flow counters.
  - Added support for Rx hardware timestamp offload.
  - Added support for device removal event.

- **Added SoftNIC PMD.**

  Added a new SoftNIC PMD. This virtual device provides applications with software fallback support for traffic management.

- **Added support for NXP DPAA Devices.**

  Added support for NXP's DPAA devices - LS104x series. This includes:

  - DPAA Bus driver
  - DPAA Mempool driver for supporting offloaded packet memory pool
  - DPAA PMD for DPAA devices

  See the *DPAA Poll Mode Driver* document for more details of this new driver.

- **Updated support for Cavium OCTEONTX Device.**

  Updated support for Cavium's OCTEONTX device (CN83xx). This includes:

  - OCTEONTX Mempool driver for supporting offloaded packet memory pool

  - OCTEONTX Ethdev PMD
  - OCTEONTX Eventdev-Ethdev Rx adapter

  See the *OCTEON TX Poll Mode driver* document for more details of this new driver.

- **Added PF support to the Netronome NFP PMD.**

  Added PF support to the Netronome NFP PMD. Previously the NFP PMD only supported VFs. PF support is just as a basic DPDK port and has no VF management yet.

  PF support comes with firmware upload support which allows the PMD to independently work from kernel netdev NFP drivers.

  NFP 4000 devices are also now supported along with previous 6000 devices.

- **Updated bnxt PMD.**

  Major enhancements include:

  - Support for Flow API
  - Support for Tx and Rx descriptor status functions

- **Added bus agnostic functions to cryptodev for PMD initialization**

  Added new PMD assist, bus independent, functions `rte_cryptodev_pmd_parse_input_args()`, `rte_cryptodev_pmd_create()` and `rte_cryptodev_pmd_destroy()` for drivers to manage creation and destruction of new device instances.

- **Updated QAT crypto PMD.**

  Added several performance enhancements:

  - Removed atomics from the internal queue pair structure.
  - Added coalesce writes to HEAD CSR on response processing.
  - Added coalesce writes to TAIL CSR on request processing.

  In addition support was added for the AES CCM algorithm.

- **Updated the AESNI MB PMD.**

  The AESNI MB PMD has been updated with additional support for:

  - The DES CBC algorithm.
  - The DES DOCSIS BPI algorithm.

  This change requires version 0.47 of the IPsec Multi-buffer library. For more details see the *AESNI Multi Buffer Crypto Poll Mode Driver* documentation.

- **Updated the OpenSSL PMD.**

  The OpenSSL PMD has been updated with additional support for:

  - The DES CBC algorithm.
  - The AES CCM algorithm.

- **Added NXP DPAA SEC crypto PMD.**

  A new `dpaa_sec` hardware based crypto PMD for NXP DPAA devices has been added. See the *NXP DPAA CAAM (DPAA_SEC)* document for more details.

- **Added MRVL crypto PMD.**

  A new crypto PMD has been added, which provides several ciphering and hashing algorithms. All cryptography operations use the MUSDK library crypto API. See the *MVSAM Crypto Poll Mode Driver* document for more details.

- **Add new benchmarking mode to dpdk-test-crypto-perf application.**

  Added a new "PMD cyclecount" benchmark mode to the `dpdk-test-crypto-perf` application to display a detailed breakdown of CPU cycles used by hardware acceleration.

- **Added the Security Offload Library.**

  Added an experimental library - `rte_security`. This provide security APIs for protocols like IPsec using inline ipsec offload to ethernet devices or full protocol offload with lookaside crypto devices.

  See the *Security Library* section of the DPDK Programmers Guide document for more information.

- **Updated the DPAA2_SEC crypto driver to support rte_security.**

  Updated the `dpaa2_sec` crypto PMD to support `rte_security` lookaside protocol offload for IPsec.

- **Updated the IXGBE ethernet driver to support rte_security.**

  Updated ixgbe ethernet PMD to support `rte_security` inline IPsec offload.

- **Updated i40e driver to support GTP-C/GTP-U.**

  Updated i40e PMD to support GTP-C/GTP-U with GTP-C/GTP-U supporting profiles which can be programmed by dynamic device personalization (DDP) process.

- **Added the i40e ethernet driver to support queue region feature.**

  This feature enable queue regions configuration for RSS in PF, so that different traffic classes or different packet classification types can be separated into different queues in different queue regions.

- **Updated ipsec-secgw application to support rte_security.**

  Updated the `ipsec-secgw` sample application to support `rte_security` actions for ipsec inline and full protocol offload using lookaside crypto offload.

- **Added IOMMU support to libvhost-user**

  Implemented device IOTLB in the Vhost-user backend, and enabled Virtio's IOMMU feature. The feature is disabled by default, and can be enabled by setting `RTE_VHOST_USER_IOMMU_SUPPORT` flag at vhost device registration time.

- **Added the Event Ethernet Adapter Library.**

  Added the Event Ethernet Adapter library. This library provides APIs for eventdev applications to configure the ethdev for eventdev packet flow.

- **Updated DPAA2 Event PMD for the Event Ethernet Adapter.**

  Added support for the eventdev ethernet adapter for DPAA2.

- **Added Membership library (rte_member).**

  Added a new data structure library called the Membership Library.

The Membership Library is an extension and generalization of a traditional filter (for example Bloom Filter) structure that has multiple usages in a wide variety of workloads and applications. In general, the Membership Library is a data structure that provides a "set-summary" and responds to set-membership queries whether a certain member belongs to a set(s).

The library provides APIs for DPDK applications to insert a new member, delete an existing member, and query the existence of a member in a given set, or a group of sets. For the case of a group of sets the library will return not only whether the element has been inserted in one of the sets but also which set it belongs to.

See the *Membership Library* documentation in the Programmers Guide, for more information.

- **Added the Generic Segmentation Offload Library.**

  Added the Generic Segmentation Offload (GSO) library to enable applications to split large packets (e.g. MTU is 64KB) into small ones (e.g. MTU is 1500B). Supported packet types are:

  - TCP/IPv4 packets.
  - VxLAN packets, which must have an outer IPv4 header, and contain an inner TCP/IPv4 packet.
  - GRE packets, which must contain an outer IPv4 header, and inner TCP/IPv4 headers.

  The GSO library doesn't check if the input packets have correct checksums, and doesn't update checksums for output packets. Additionally, the GSO library doesn't process IP fragmented packets.

- **Added the Flow Classification Library.**

  Added an experimental Flow Classification library to provide APIs for DPDK applications to classify an input packet by matching it against a set of flow rules. It uses the `librte_table` API to manage the flow rules.

19.11.2 Resolved Issues

- **Service core fails to call service callback due to atomic lock**

  In a specific configuration of multi-thread unsafe services and service cores, a service core previously did not correctly release the atomic lock on the service. This would result in the cores polling the service, but it looked like another thread was executing the service callback. The logic for atomic locking of the services has been fixed and refactored for readability.

19.11.3 API Changes

- **Ethdev device name length increased.**

  The size of internal device name has been increased to 64 characters to allow for storing longer bus specific names.

- **Removed the Ethdev RTE_ETH_DEV_DETACHABLE flag.**

  Removed the Ethdev `RTE_ETH_DEV_DETACHABLE` flag. This flag is not required anymore, with the new hotplug implementation. It has been removed from the ether library. Its semantics are now expressed at the bus and PMD level.

- **Service cores API updated for usability**

  The service cores API has been changed, removing pointers from the API where possible, and instead using integer IDs to identify each service. This simplifies application code, aids debugging, and provides better encapsulation. A summary of the main changes made is as follows:

  - Services identified by ID not by `rte_service_spec` pointer
  - Reduced API surface by using `set` functions instead of enable/disable
  - Reworked `rte_service_register` to provide the service ID to registrar
  - Reworked start and stop APIs into `rte_service_runstate_set`
  - Added API to set runstate of service implementation to indicate readiness

- **The following changes have been made in the mempool library**

  - Moved `flags` datatype from `int` to `unsigned int` for `rte_mempool`.
  - Removed `__rte_unused int flag` param from `rte_mempool_generic_put` and `rte_mempool_generic_get` API.
  - Added `flags` param in `rte_mempool_xmem_size` and `rte_mempool_xmem_usage`.
  - `rte_mem_phy2mch` was used in Xen dom0 to obtain the physical address; remove this API as Xen dom0 support was removed.

- **Added IOVA aliases related to physical address handling.**

  Some data types, structure members and functions related to physical address handling are deprecated and have new aliases with IOVA wording. For example:

  - `phys_addr_t` can be often replaced by `rte_iova_t` of same size.
  - `RTE_BAD_PHYS_ADDR` is often replaced by `RTE_BAD_IOVA` of same value.
  - `rte_memseg.phys_addr` is aliased with `rte_memseg.iova_addr`.
  - `rte_mem_virt2phy()` can often be replaced by `rte_mem_virt2iova`.
  - `rte_malloc_virt2phy` is aliased with `rte_malloc_virt2iova`.
  - `rte_memzone.phys_addr` is aliased with `rte_memzone.iova`.
  - `rte_mempool_objhdr.physaddr` is aliased with `rte_mempool_objhdr.iova`.
  - `rte_mempool_memhdr.phys_addr` is aliased with `rte_mempool_memhdr.iova`.
  - `rte_mempool_virt2phy()` can be replaced by `rte_mempool_virt2iova()`.
  - `rte_mempool_populate_phys*()` are aliased with `rte_mempool_populate_iova*()`
  - `rte_mbuf.buf_physaddr` is aliased with `rte_mbuf.buf_iova`.
  - `rte_mbuf_data_dma_addr*()` are aliased with `rte_mbuf_data_iova*()`.
  - `rte_pktmbuf_mtophys*` are aliased with `rte_pktmbuf_iova*()`.

- **PCI bus API moved outside of the EAL**

  The PCI bus previously implemented within the EAL has been moved. A first part has been added as an RTE library providing PCI helpers to parse device locations or other such utilities. A second part consisting of the actual bus driver has been moved to its proper subdirectory, without changing its functionalities.

As such, several PCI-related functions are not exposed by the EAL anymore:

- rte_pci_detach
- rte_pci_dump
- rte_pci_ioport_map
- rte_pci_ioport_read
- rte_pci_ioport_unmap
- rte_pci_ioport_write
- rte_pci_map_device
- rte_pci_probe
- rte_pci_probe_one
- rte_pci_read_config
- rte_pci_register
- rte_pci_scan
- rte_pci_unmap_device
- rte_pci_unregister
- rte_pci_write_config

These functions are made available either as part of librte_pci or librte_bus_pci.

- **Moved vdev bus APIs outside of the EAL**

  Moved the following APIs from librte_eal to librte_bus_vdev:

  - rte_vdev_init
  - rte_vdev_register
  - rte_vdev_uninit
  - rte_vdev_unregister

- **Add return value to stats_get dev op API**

  The stats_get dev op API return value has been changed to be int. In this way PMDs can return an error value in case of failure at stats getting process time.

- **Modified the rte_cryptodev_allocate_driver function.**

  Modified the rte_cryptodev_allocate_driver() function in the cryptodev library. An extra parameter struct cryptodev_driver *crypto_drv has been added.

- **Removed virtual device bus specific functions from librte_cryptodev.**

  The functions rte_cryptodev_vdev_parse_init_params() and rte_cryptodev_vdev_pmd_init() have been removed from librte_cryptodev and have been replaced by non bus specific functions rte_cryptodev_pmd_parse_input_args() and rte_cryptodev_pmd_create().

  The rte_cryptodev_create_vdev() function was removed to avoid the dependency on vdev in librte_cryptodev; instead, users can call rte_vdev_init() directly.

- **Removed PCI device bus specific functions from librte_cryptodev.**

  The functions `rte_cryptodev_pci_generic_probe()` and `rte_cryptodev_pci_generic_remove()` have been removed from librte_cryptodev and have been replaced by non bus specific functions `rte_cryptodev_pmd_create()` and `rte_cryptodev_pmd_destroy()`.

- **Removed deprecated functions to manage log level or type.**

  The functions `rte_set_log_level()`, `rte_get_log_level()`, `rte_set_log_type()` and `rte_get_log_type()` have been removed.

  They are respectively replaced by `rte_log_set_global_level()`, `rte_log_get_global_level()`, `rte_log_set_level()` and `rte_log_get_level()`.

- **Removed mbuf flags PKT_RX_VLAN_PKT and PKT_RX_QINQ_PKT.**

  The `mbuf` flags `PKT_RX_VLAN_PKT` and `PKT_RX_QINQ_PKT` have been removed since their behavior was not properly described.

- **Added mbuf flags PKT_RX_VLAN and PKT_RX_QINQ.**

  Two `mbuf` flags have been added to indicate that the VLAN identifier has been saved in the `mbuf` structure. For instance:

  - If VLAN is not stripped and TCI is saved: `PKT_RX_VLAN`
  - If VLAN is stripped and TCI is saved: `PKT_RX_VLAN | PKT_RX_VLAN_STRIPPED`

- **Modified the vlan_offload_set_t function prototype in the ethdev library.**

  Modified the `vlan_offload_set_t` function prototype in the ethdev library. The return value has been changed from `void` to `int` so the caller can determine whether the backing device supports the operation or if the operation was successfully performed.

19.11.4 ABI Changes

- **Extended port_id range.**

  The size of the field `port_id` in the `rte_eth_dev_data` structure has changed, as described in the *New Features* section above.

- **New parameter added to rte_eth_dev.**

  A new parameter `security_ctx` has been added to `rte_eth_dev` to support security operations like IPsec inline.

- **New parameter added to rte_cryptodev.**

  A new parameter `security_ctx` has been added to `rte_cryptodev` to support security operations like lookaside crypto.

19.11.5 Removed Items

- Xen dom0 in EAL has been removed, as well as the xenvirt PMD and vhost_xen.
- The crypto performance unit tests have been removed, replaced by the `dpdk-test-crypto-perf` application.

19.11.6 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
+ librte_bitratestats.so.2
+ librte_bus_dpaa.so.1
+ librte_bus_fslmc.so.1
+ librte_bus_pci.so.1
+ librte_bus_vdev.so.1
  librte_cfgfile.so.2
  librte_cmdline.so.2
+ librte_cryptodev.so.4
  librte_distributor.so.1
+ librte_eal.so.6
+ librte_ethdev.so.8
+ librte_eventdev.so.3
+ librte_flow_classify.so.1
  librte_gro.so.1
+ librte_gso.so.1
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_latencystats.so.1
  librte_lpm.so.2
  librte_mbuf.so.3
+ librte_mempool.so.3
  librte_meter.so.1
  librte_metrics.so.1
  librte_net.so.1
+ librte_pci.so.1
+ librte_pdump.so.2
  librte_pipeline.so.3
+ librte_pmd_bnxt.so.2
+ librte_pmd_bond.so.2
+ librte_pmd_i40e.so.2
+ librte_pmd_ixgbe.so.2
  librte_pmd_ring.so.2
+ librte_pmd_softnic.so.1
+ librte_pmd_vhost.so.2
  librte_port.so.3
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
+ librte_security.so.1
+ librte_table.so.3
  librte_timer.so.1
  librte_vhost.so.3
```

19.11.7 Tested Platforms

- Intel(R) platforms with Intel(R) NICs combinations
  - CPU
    - Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
    - Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
    - Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
    - Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-2658 v2 @ 2.40GHz
    - Intel(R) Xeon(R) CPU E5-2658 v3 @ 2.20GHz
  - OS:
    - CentOS 7.2
    - Fedora 25
    - Fedora 26
    - FreeBSD 11
    - Red Hat Enterprise Linux Server release 7.3
    - SUSE Enterprise Linux 12
    - Wind River Linux 8
    - Ubuntu 16.04
    - Ubuntu 16.10
  - NICs:
    - Intel(R) 82599ES 10 Gigabit Ethernet Controller
      - Firmware version: 0x61bf0001
      - Device id (pf/vf): 8086:10fb / 8086:10ed
      - Driver version: 5.2.3 (ixgbe)
    - Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      - Firmware version: 0x800003e7
      - Device id (pf/vf): 8086:15ad / 8086:15a8
      - Driver version: 4.4.6 (ixgbe)
    - Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
      - Firmware version: 6.01 0x80003205
      - Device id (pf/vf): 8086:1572 / 8086:154c

            - Driver version: 2.1.26 (i40e)
        * Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
            - Firmware version: 6.01 0x80003204
            - Device id (pf/vf): 8086:1572 / 8086:154c
            - Driver version: 2.1.26 (i40e)
        * Intel(R) Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
            - Firmware version: 6.01 0x80003221
            - Device id (pf/vf): 8086:158b
            - Driver version: 2.1.26 (i40e)
        * Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
            - Firmware version: 6.01 0x8000321c
            - Device id (pf/vf): 8086:1583 / 8086:154c
            - Driver version: 2.1.26 (i40e)
        * Intel(R) Corporation I350 Gigabit Network Connection
            - Firmware version: 1.63, 0x80000dda
            - Device id (pf/vf): 8086:1521 / 8086:1520
            - Driver version: 5.3.0-k (igb)
- Intel(R) platforms with Mellanox(R) NICs combinations
    - Platform details:
        * Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
        * Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
        * Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
        * Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
        * Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
        * Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
    - OS:
        * Red Hat Enterprise Linux Server release 7.3 (Maipo)
        * Red Hat Enterprise Linux Server release 7.2 (Maipo)
        * Ubuntu 16.10
        * Ubuntu 16.04
        * Ubuntu 14.04
    - MLNX_OFED: 4.2-1.0.0.0
    - NICs:
        * Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)

- Host interface: PCI Express 3.0 x8
- Device ID: 15b3:1007
- Firmware version: 2.42.5000

* Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000
* Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000
* Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000
* Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000
* Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000
* Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000
* Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.21.1000
* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013

            - Firmware version: 12.21.1000
        - Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1013
            - Firmware version: 12.21.1000
        - Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1013
            - Firmware version: 12.21.1000
        - Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1013
            - Firmware version: 12.21.1000
        - Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.21.1000
        - Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1015
            - Firmware version: 14.21.1000
        - Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
            - Host interface: PCI Express 3.0 x16
            - Device ID: 15b3:1017
            - Firmware version: 16.21.1000
        - Mellanox(R) ConnectX-5 Ex EN 100G MCX516A-CDAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1019
            - Firmware version: 16.21.1000
- ARM platforms with Mellanox(R) NICs combinations
    - Platform details:
        - Qualcomm ARM 1.1 2500MHz
    - OS:
        - Ubuntu 16.04

- MLNX_OFED: 4.2-1.0.0.0
- NICs:
  - Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1015
    - Firmware version: 14.21.1000
  - Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1017
    - Firmware version: 16.21.1000

19.12 DPDK Release 17.08

19.12.1 New Features

- **Increase minimum x86 ISA version to SSE4.2.**

  Starting with version 17.08, DPDK requires SSE4.2 to run on x86. Previous versions required SSE3.

- **Added Service Core functionality.**

  The service core functionality added to EAL allows DPDK to run services such as software PMDs on lcores without the application manually running them. The service core infrastructure allows flexibility of running multiple services on the same service lcore, and provides the application with powerful APIs to configure the mapping from service lcores to services.

- **Added Generic Receive Offload API.**

  Added Generic Receive Offload (GRO) API support to reassemble TCP/IPv4 packets. The GRO API assumes all input packets have the correct checksums. GRO API doesn't update checksums for merged packets. If input packets are IP fragmented, the GRO API assumes they are complete packets (i.e. with L4 headers).

- **Added Fail-Safe PMD**

  Added the new Fail-Safe PMD. This virtual device allows applications to support seamless hotplug of devices. See the *Fail-safe poll mode driver library* guide for more details about this driver.

- **Added support for generic flow API (rte_flow) on igb NICs.**

  This API provides a generic means of configuring hardware to match specific ingress or egress traffic, altering its behavior and querying related counters according to any number of user-defined rules.

  Added generic flow API support for Ethernet, IPv4, UDP, TCP and RAW pattern items with QUEUE actions. There are four types of filter support for this feature on igb.

- **Added support for generic flow API (rte_flow) on enic.**

Added flow API support for outer Ethernet, VLAN, IPv4, IPv6, UDP, TCP, SCTP, VxLAN and inner Ethernet, VLAN, IPv4, IPv6, UDP and TCP pattern items with QUEUE, MARK, FLAG and VOID actions for ingress traffic.

- **Added support for Chelsio T6 family of adapters**

  The CXGBE PMD was updated to run Chelsio T6 family of adapters.

- **Added latency and performance improvements for cxgbe**

  the Tx and Rx path in cxgbe were reworked to improve performance. In addition the latency was reduced for slow traffic.

- **Updated the bnxt PMD.**

  Updated the bnxt PMD. The major enhancements include:

  - Support MTU modification.
  - Add support for LRO.
  - Add support for VLAN filter and strip functionality.
  - Additional enhancements to add support for more dev_ops.
  - Added PMD specific APIs mainly to control VF from PF.
  - Update HWRM version to 1.7.7

- **Added support for Rx interrupts on mlx4 driver.**

  Rx queues can be now be armed with an interrupt which will trigger on the next packet arrival.

- **Updated mlx5 driver.**

  Updated the mlx5 driver including the following changes:

  - Added vectorized Rx/Tx burst for x86.
  - Added support for isolated mode from flow API.
  - Reworked the flow drop action to implement in hardware classifier.
  - Improved Rx interrupts management.

- **Updated szedata2 PMD.**

  Added support for firmware with multiple Ethernet ports per physical port.

- **Updated dpaa2 PMD.**

  Updated dpaa2 PMD. Major enhancements include:

  - Added support for MAC Filter configuration.
  - Added support for Segmented Buffers.
  - Added support for VLAN filter and strip functionality.
  - Additional enhancements to add support for more dev_ops.
  - Optimized the packet receive path

- **Reorganized the symmetric crypto operation structure.**

  The crypto operation (`rte_crypto_sym_op`) has been reorganized as follows:

  - Removed the `rte_crypto_sym_op_sess_type` field.
  - Replaced the pointer and physical address of IV with offset from the start of the crypto operation.
  - Moved length and offset of cipher IV to `rte_crypto_cipher_xform`.
  - Removed "Additional Authentication Data" (AAD) length.
  - Removed digest length.
  - Removed AAD pointer and physical address from `auth` structure.
  - Added `aead` structure, containing parameters for AEAD algorithms.

- **Reorganized the crypto operation structure.**

  The crypto operation (`rte_crypto_op`) has been reorganized as follows:

  - Added the `rte_crypto_op_sess_type` field.
  - The enumerations `rte_crypto_op_status` and `rte_crypto_op_type` have been modified to be `uint8_t` values.
  - Removed the field `opaque_data`.
  - Pointer to `rte_crypto_sym_op` has been replaced with a zero length array.

- **Reorganized the crypto symmetric session structure.**

  The crypto symmetric session structure (`rte_cryptodev_sym_session`) has been reorganized as follows:

  - The `dev_id` field has been removed.
  - The `driver_id` field has been removed.
  - The mempool pointer `mp` has been removed.
  - Replaced `private` marker with array of pointers to private data sessions `sess_private_data`.

- **Updated cryptodev library.**

  - Added AEAD algorithm specific functions and structures, so it is not necessary to use a combination of cipher and authentication structures anymore.
  - Added helper functions for crypto device driver identification.
  - Added support for multi-device sessions, so a single session can be used in multiple drivers.
  - Added functions to initialize and free individual driver private data with the same session.

- **Updated dpaa2_sec crypto PMD.**

  Added support for AES-GCM and AES-CTR.

- **Updated the AESNI MB PMD.**

  The AESNI MB PMD has been updated with additional support for:

  - 12-byte IV on AES Counter Mode, apart from the previous 16-byte IV.

- **Updated the AES-NI GCM PMD.**

  The AES-NI GCM PMD was migrated from the ISA-L library to the Multi Buffer library, as the latter library has Scatter Gather List support now. The migration entailed adding additional support for 192-bit keys.

- **Updated the Cryptodev Scheduler PMD.**

  Added a multicore based distribution mode, which distributes the enqueued crypto operations among several slaves, running on different logical cores.

- **Added NXP DPAA2 Eventdev PMD.**

  Added the new dpaa2 eventdev driver for NXP DPAA2 devices. See the "Event Device Drivers" document for more details on this new driver.

- **Added dpdk-test-eventdev test application.**

  The dpdk-test-eventdev tool is a Data Plane Development Kit (DPDK) application that allows exercising various eventdev use cases. This application has a generic framework to add new eventdev based test cases to verify functionality and measure the performance parameters of DPDK eventdev devices.

19.12.2 Known Issues

- **Starting with version 17.08, libnuma is required to build DPDK.**

19.12.3 API Changes

- **Modified the _rte_eth_dev_callback_process function in the ethdev library.**

  The function `_rte_eth_dev_callback_process()` has been modified. The return value has been changed from void to int and an extra parameter `void *ret_param` has been added.

- **Moved bypass functions from the rte_ethdev library to ixgbe PMD**
  - The following rte_ethdev library functions were removed:
    - `rte_eth_dev_bypass_event_show()`
    - `rte_eth_dev_bypass_event_store()`
    - `rte_eth_dev_bypass_init()`
    - `rte_eth_dev_bypass_state_set()`
    - `rte_eth_dev_bypass_state_show()`
    - `rte_eth_dev_bypass_ver_show()`
    - `rte_eth_dev_bypass_wd_reset()`
    - `rte_eth_dev_bypass_wd_timeout_show()`
    - `rte_eth_dev_wd_timeout_store()`
  - The following ixgbe PMD functions were added:
    - `rte_pmd_ixgbe_bypass_event_show()`
    - `rte_pmd_ixgbe_bypass_event_store()`

    * rte_pmd_ixgbe_bypass_init()
    * rte_pmd_ixgbe_bypass_state_set()
    * rte_pmd_ixgbe_bypass_state_show()
    * rte_pmd_ixgbe_bypass_ver_show()
    * rte_pmd_ixgbe_bypass_wd_reset()
    * rte_pmd_ixgbe_bypass_wd_timeout_show()
    * rte_pmd_ixgbe_bypass_wd_timeout_store()

- **Reworked rte_cryptodev library.**

  The rte_cryptodev library has been reworked and updated. The following changes have been made to it:

  - The crypto device type enumeration has been removed from cryptodev library.
  - The function `rte_crypto_count_devtype()` has been removed, and replaced by the new function `rte_crypto_count_by_driver()`.
  - Moved crypto device driver names definitions to the particular PMDs. These names are not public anymore.
  - The `rte_cryptodev_configure()` function does not create the session mempool for the device anymore.
  - The `rte_cryptodev_queue_pair_attach_sym_session()` and `rte_cryptodev_queue_pair_dettach_sym_session()` functions require the new parameter `device id`.
  - Parameters of `rte_cryptodev_sym_session_create()` were modified to accept `mempool`, instead of `device id` and `rte_crypto_sym_xform`.
  - Removed `device id` parameter from `rte_cryptodev_sym_session_free()`.
  - Added a new field `session_pool` to `rte_cryptodev_queue_pair_setup()`.
  - Removed `aad_size` parameter from `rte_cryptodev_sym_capability_check_auth()`.
  - Added `iv_size` parameter to `rte_cryptodev_sym_capability_check_auth()`.
  - Removed `RTE_CRYPTO_OP_STATUS_ENQUEUED` from enum `rte_crypto_op_status`.

19.12.4 ABI Changes

- Changed type of `domain` field in `rte_pci_addr` to `uint32_t` to follow the PCI standard.
- Added new `rte_bus` experimental APIs available as operators within the `rte_bus` structure.
- Made `rte_devargs` structure internal device representation generic to prepare for a bus-agnostic EAL.
- **Reorganized the crypto operation structures.**

  Some fields have been modified in the `rte_crypto_op` and `rte_crypto_sym_op` structures, as described in the *New Features* section.

- **Reorganized the crypto symmetric session structure.**

  Some fields have been modified in the `rte_cryptodev_sym_session` structure, as described in the *New Features* section.

- **Reorganized the rte_crypto_sym_cipher_xform structure.**
  - Added cipher IV length and offset parameters.
  - Changed field size of key length from `size_t` to `uint16_t`.
- **Reorganized the rte_crypto_sym_auth_xform structure.**
  - Added authentication IV length and offset parameters.
  - Changed field size of AAD length from `uint32_t` to `uint16_t`.
  - Changed field size of digest length from `uint32_t` to `uint16_t`.
  - Removed AAD length.
  - Changed field size of key length from `size_t` to `uint16_t`.
- Replaced `dev_type` enumeration with `uint8_t driver_id` in `rte_cryptodev_info` and `rte_cryptodev` structures.
- Removed `session_mp` from `rte_cryptodev_config`.

19.12.5 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_bitratestats.so.1
  librte_cfgfile.so.2
  librte_cmdline.so.2
+ librte_cryptodev.so.3
  librte_distributor.so.1
+ librte_eal.so.5
+ librte_ethdev.so.7
+ librte_eventdev.so.2
+ librte_gro.so.1
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_latencystats.so.1
  librte_lpm.so.2
  librte_mbuf.so.3
  librte_mempool.so.2
  librte_meter.so.1
  librte_metrics.so.1
  librte_net.so.1
  librte_pdump.so.1
  librte_pipeline.so.3
  librte_pmd_bond.so.1
  librte_pmd_ring.so.2
  librte_port.so.3
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
```

(continues on next page)

(continued from previous page)

```
librte_sched.so.1
librte_table.so.2
librte_timer.so.1
librte_vhost.so.3
```

19.12.6 Tested Platforms

- Intel(R) platforms with Mellanox(R) NICs combinations
  - Platform details:
    - Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
    - Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
  - OS:
    - Red Hat Enterprise Linux Server release 7.3 (Maipo)
    - Red Hat Enterprise Linux Server release 7.2 (Maipo)
    - Ubuntu 16.10
    - Ubuntu 16.04
    - Ubuntu 14.04
  - MLNX_OFED: 4.1-1.0.2.0
  - NICs:
    - Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1007
      - Firmware version: 2.40.5030
    - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.2000
    - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.2000
    - Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013

- Firmware version: 12.18.2000

* Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)

        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1015
        - Firmware version: 14.18.2000
      * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1015
        - Firmware version: 14.18.2000
      * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1017
        - Firmware version: 16.19.1200
      * Mellanox(R) ConnectX-5 Ex EN 100G MCX516A-CDAT (2x100G)
        - Host interface: PCI Express 4.0 x16
        - Device ID: 15b3:1019
        - Firmware version: 16.19.1200
- Intel(R) platforms with Intel(R) NICs combinations
  - CPU
    * Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
    * Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
    * Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
    * Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
    * Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    * Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
    * Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
    * Intel(R) Xeon(R) CPU E5-2658 v2 @ 2.40GHz
    * Intel(R) Xeon(R) CPU E5-2658 v3 @ 2.20GHz
  - OS:
    * CentOS 7.2
    * Fedora 25
    * FreeBSD 11
    * Red Hat Enterprise Linux Server release 7.3
    * SUSE Enterprise Linux 12
    * Wind River Linux 8
    * Ubuntu 16.04

    - Ubuntu 16.10

- NICs:
    - Intel(R) 82599ES 10 Gigabit Ethernet Controller
        - Firmware version: 0x61bf0001
        - Device id (pf/vf): 8086:10fb / 8086:10ed
        - Driver version: 4.0.1-k (ixgbe)
    - Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
        - Firmware version: 0x800001cf
        - Device id (pf/vf): 8086:15ad / 8086:15a8
        - Driver version: 4.2.5 (ixgbe)
    - Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
        - Firmware version: 6.01 0x80003205
        - Device id (pf/vf): 8086:1572 / 8086:154c
        - Driver version: 2.0.19 (i40e)
    - Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
        - Firmware version: 6.01 0x80003204
        - Device id (pf/vf): 8086:1572 / 8086:154c
        - Driver version: 2.0.19 (i40e)
    - Intel(R) Ethernet Converged Network Adapter XXV710-DA2 (2x25G)
        - Firmware version: 6.01 0x80003221
        - Device id (pf/vf): 8086:158b
        - Driver version: 2.0.19 (i40e)
    - Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
        - Firmware version: 6.01 0x8000321c
        - Device id (pf/vf): 8086:1583 / 8086:154c
        - Driver version: 2.0.19 (i40e)
    - Intel(R) Corporation I350 Gigabit Network Connection
        - Firmware version: 1.48, 0x800006e7
        - Device id (pf/vf): 8086:1521 / 8086:1520
        - Driver version: 5.2.13-k (igb)

19.13 DPDK Release 17.05

19.13.1 New Features

- **Reorganized mbuf structure.**

  The mbuf structure has been reorganized as follows:

  - Align fields to facilitate the writing of `data_off`, `refcnt`, and `nb_segs` in one operation.
  - Use 2 bytes for port and number of segments.
  - Move the sequence number to the second cache line.
  - Add a timestamp field.
  - Set default value for `refcnt`, `next` and `nb_segs` at mbuf free.

- **Added mbuf raw free API.**

  Moved `rte_mbuf_raw_free()` and `rte_pktmbuf_prefree_seg()` functions to the public API.

- **Added free Tx mbuf on demand API.**

  Added a new function `rte_eth_tx_done_cleanup()` which allows an application to request the driver to release mbufs that are no longer in use from a Tx ring, independent of whether or not the `tx_rs_thresh` has been crossed.

- **Added device removal interrupt.**

  Added a new ethdev event `RTE_ETH_DEV_INTR_RMV` to signify the sudden removal of a device. This event can be advertised by PCI drivers and enabled accordingly.

- **Added EAL dynamic log framework.**

  Added new APIs to dynamically register named log types, and control the level of each type independently.

- **Added descriptor status ethdev API.**

  Added a new API to get the status of a descriptor.

  For Rx, it is almost similar to the `rx_descriptor_done` API, except it differentiates descriptors which are held by the driver and not returned to the hardware. For Tx, it is a new API.

- **Increased number of next hops for LPM IPv6 to 2^21.**

  The next_hop field has been extended from 8 bits to 21 bits for IPv6.

- **Added VFIO hotplug support.**

  Added hotplug support for VFIO in addition to the existing UIO support.

- **Added PowerPC support to pci probing for vfio-pci devices.**

  Enabled sPAPR IOMMU based pci probing for vfio-pci devices.

- **Kept consistent PMD batching behavior.**

  Removed the limit of fm10k/i40e/ixgbe Tx burst size and vhost Rx/Tx burst size in order to support the same policy of "make an best effort to Rx/Tx pkts" for PMDs.

- **Updated the ixgbe base driver.**

  Updated the ixgbe base driver, including the following changes:

  - Add link block check for KR.
  - Complete HW initialization even if SFP is not present.
  - Add VF xcast promiscuous mode.

- **Added PowerPC support for i40e and its vector PMD.**

  Enabled i40e PMD and its vector PMD by default in PowerPC.

- **Added VF max bandwidth setting in i40e.**

  Enabled capability to set the max bandwidth for a VF in i40e.

- **Added VF TC min and max bandwidth setting in i40e.**

  Enabled capability to set the min and max allocated bandwidth for a TC on a VF in i40.

- **Added TC strict priority mode setting on i40e.**

  There are 2 Tx scheduling modes supported for TCs by i40e HW: round robin mode and strict priority mode. By default the round robin mode is used. It is now possible to change the Tx scheduling mode for a TC. This is a global setting on a physical port.

- **Added i40e dynamic device personalization support.**

  - Added dynamic device personalization processing to i40e firmware.

- **Updated i40e driver to support MPLSoUDP/MPLSoGRE.**

  Updated i40e PMD to support MPLSoUDP/MPLSoGRE with MPLSoUDP/MPLSoGRE supporting profiles which can be programmed by dynamic device personalization (DDP) process.

- **Added Cloud Filter for QinQ steering to i40e.**

  - Added a QinQ cloud filter on the i40e PMD, for steering traffic to a VM using both VLAN tags. Note, this feature is not supported in Vector Mode.

- **Updated mlx5 PMD.**

  Updated the mlx5 driver, including the following changes:

  - Added Generic flow API support for classification according to ether type.
  - Extended Generic flow API support for classification of IPv6 flow according to Vtc flow, Protocol and Hop limit.
  - Added Generic flow API support for FLAG action.
  - Added Generic flow API support for RSS action.
  - Added support for TSO for non-tunneled and VXLAN packets.
  - Added support for hardware Tx checksum offloads for VXLAN packets.
  - Added support for user space Rx interrupt mode.
  - Improved ConnectX-5 single core and maximum performance.

- **Updated mlx4 PMD.**

  Updated the mlx4 driver, including the following changes:

  - Added support for Generic flow API basic flow items and actions.
  - Added support for device removal event.

- **Updated the sfc_efx driver.**
  - Added Generic Flow API support for Ethernet, VLAN, IPv4, IPv6, UDP and TCP pattern items with QUEUE action for ingress traffic.
  - Added support for virtual functions (VFs).

- **Added LiquidIO network PMD.**

  Added poll mode driver support for Cavium LiquidIO II server adapter VFs.

- **Added Atomic Rules Arkville PMD.**

  Added a new poll mode driver for the Arkville family of devices from Atomic Rules. The net/ark PMD supports line-rate agnostic, multi-queue data movement on Arkville core FPGA instances.

- **Added support for NXP DPAA2 - FSLMC bus.**

  Added the new bus "fslmc" driver for NXP DPAA2 devices. See the "Network Interface Controller Drivers" document for more details of this new driver.

- **Added support for NXP DPAA2 Network PMD.**

  Added the new "dpaa2" net driver for NXP DPAA2 devices. See the "Network Interface Controller Drivers" document for more details of this new driver.

- **Added support for the Wind River Systems AVP PMD.**

  Added a new networking driver for the AVP device type. Theses devices are specific to the Wind River Systems virtualization platforms.

- **Added vmxnet3 version 3 support.**

  Added support for vmxnet3 version 3 which includes several performance enhancements such as configurable Tx data ring, Receive Data Ring, and the ability to register memory regions.

- **Updated the TAP driver.**

  Updated the TAP PMD to:

  - Support MTU modification.
  - Support packet type for Rx.
  - Support segmented packets on Rx and Tx.
  - Speed up Rx on TAP when no packets are available.
  - Support capturing traffic from another netdevice.
  - Dynamically change link status when the underlying interface state changes.
  - Added Generic Flow API support for Ethernet, VLAN, IPv4, IPv6, UDP and TCP pattern items with DROP, QUEUE and PASSTHRU actions for ingress traffic.

- **Added MTU feature support to Virtio and Vhost.**

  Implemented new Virtio MTU feature in Vhost and Virtio:

  - Add `rte_vhost_mtu_get()` API to Vhost library.
  - Enable Vhost PMD's MTU get feature.

  - Get max MTU value from host in Virtio PMD

- **Added interrupt mode support for virtio-user.**

  Implemented Rxq interrupt mode and LSC support for virtio-user as a virtual device. Supported cases:

  - Rxq interrupt for virtio-user + vhost-user as the backend.
  - Rxq interrupt for virtio-user + vhost-kernel as the backend.
  - LSC interrupt for virtio-user + vhost-user as the backend.

- **Added event driven programming model library (rte_eventdev).**

  This API introduces an event driven programming model.

  In a polling model, lcores poll ethdev ports and associated Rx queues directly to look for a packet. By contrast in an event driven model, lcores call the scheduler that selects packets for them based on programmer-specified criteria. The Eventdev library adds support for an event driven programming model, which offers applications automatic multicore scaling, dynamic load balancing, pipelining, packet ingress order maintenance and synchronization services to simplify application packet processing.

  By introducing an event driven programming model, DPDK can support both polling and event driven programming models for packet processing, and applications are free to choose whatever model (or combination of the two) best suits their needs.

- **Added Software Eventdev PMD.**

  Added support for the software eventdev PMD. The software eventdev is a software based scheduler device that implements the eventdev API. This PMD allows an application to configure a pipeline using the eventdev library, and run the scheduling workload on a CPU core.

- **Added Cavium OCTEONTX Eventdev PMD.**

  Added the new octeontx ssovf eventdev driver for OCTEONTX devices. See the "Event Device Drivers" document for more details on this new driver.

- **Added information metrics library.**

  Added a library that allows information metrics to be added and updated by producers, typically other libraries, for later retrieval by consumers such as applications. It is intended to provide a reporting mechanism that is independent of other libraries such as ethdev.

- **Added bit-rate calculation library.**

  Added a library that can be used to calculate device bit-rates. Calculated bitrates are reported using the metrics library.

- **Added latency stats library.**

  Added a library that measures packet latency. The collected statistics are jitter and latency. For latency the minimum, average, and maximum is measured.

- **Added NXP DPAA2 SEC crypto PMD.**

  A new "dpaa2_sec" hardware based crypto PMD for NXP DPAA2 devices has been added. See the "Crypto Device Drivers" document for more details on this driver.

- **Updated the Cryptodev Scheduler PMD.**

  - Added a packet-size based distribution mode, which distributes the enqueued crypto operations among two slaves, based on their data lengths.
  - Added fail-over scheduling mode, which enqueues crypto operations to a primary slave first. Then, any operation that cannot be enqueued is enqueued to a secondary slave.
  - Added mode specific option support, so each scheduling mode can now be configured individually by the new API.

- **Updated the QAT PMD.**

  The QAT PMD has been updated with additional support for:

  - AES DOCSIS BPI algorithm.
  - DES DOCSIS BPI algorithm.
  - ZUC EEA3/EIA3 algorithms.

- **Updated the AESNI MB PMD.**

  The AESNI MB PMD has been updated with additional support for:

  - AES DOCSIS BPI algorithm.

- **Updated the OpenSSL PMD.**

  The OpenSSL PMD has been updated with additional support for:

  - DES DOCSIS BPI algorithm.

19.13.2 Resolved Issues

- **l2fwd-keepalive: Fixed unclean shutdowns.**

  Added clean shutdown to l2fwd-keepalive so that it can free up stale resources used for inter-process communication.

19.13.3 Known Issues

- **LSC interrupt doesn't work for virtio-user + vhost-kernel.**

  LSC interrupt cannot be detected when setting the backend, tap device, up/down as we fail to find a way to monitor such event.

19.13.4 API Changes

- The LPM `next_hop` field is extended from 8 bits to 21 bits for IPv6 while keeping ABI compatibility.
- **Reworked rte_ring library.**

  The rte_ring library has been reworked and updated. The following changes have been made to it:

  - Removed the build-time setting `CONFIG_RTE_RING_SPLIT_PROD_CONS`.
  - Removed the build-time setting `CONFIG_RTE_LIBRTE_RING_DEBUG`.
  - Removed the build-time setting `CONFIG_RTE_RING_PAUSE_REP_COUNT`.

  - Removed the function `rte_ring_set_water_mark` as part of a general removal of watermarks support in the library.
  - Added an extra parameter to the burst/bulk enqueue functions to return the number of free spaces in the ring after enqueue. This can be used by an application to implement its own watermark functionality.
  - Added an extra parameter to the burst/bulk dequeue functions to return the number elements remaining in the ring after dequeue.
  - Changed the return value of the enqueue and dequeue bulk functions to match that of the burst equivalents. In all cases, ring functions which operate on multiple packets now return the number of elements enqueued or dequeued, as appropriate. The updated functions are:
    * `rte_ring_mp_enqueue_bulk`
    * `rte_ring_sp_enqueue_bulk`
    * `rte_ring_enqueue_bulk`
    * `rte_ring_mc_dequeue_bulk`
    * `rte_ring_sc_dequeue_bulk`
    * `rte_ring_dequeue_bulk`

    NOTE: the above functions all have different parameters as well as different return values, due to the other listed changes above. This means that all instances of the functions in existing code will be flagged by the compiler. The return value usage should be checked while fixing the compiler error due to the extra parameter.

- **Reworked rte_vhost library.**

  The rte_vhost library has been reworked to make it generic enough so that the user could build other vhost-user drivers on top of it. To achieve this the following changes have been made:

  - The following vhost-pmd APIs are removed:
    * `rte_eth_vhost_feature_disable`
    * `rte_eth_vhost_feature_enable`
    * `rte_eth_vhost_feature_get`
  - The vhost API `rte_vhost_driver_callback_register(ops)` is reworked to be per vhost-user socket file. Thus, it takes one more argument: `rte_vhost_driver_callback_register(path, ops)`.
  - The vhost API `rte_vhost_get_queue_num` is deprecated, instead, `rte_vhost_get_vring_num` should be used.
  - The following macros are removed in `rte_virtio_net.h`
    * `VIRTIO_RXQ`
    * `VIRTIO_TXQ`
    * `VIRTIO_QNUM`
  - The following net specific header files are removed in `rte_virtio_net.h`
    * `linux/virtio_net.h`
    * `sys/socket.h`

    * linux/if.h
    * rte_ether.h
  - The vhost struct virtio_net_device_ops is renamed to vhost_device_ops
  - The vhost API rte_vhost_driver_session_start is removed. Instead, rte_vhost_driver_start should be used, and there is no need to create a thread to call it.
  - The vhost public header file rte_virtio_net.h is renamed to rte_vhost.h

19.13.5 ABI Changes

- **Reorganized the mbuf structure.**

  The order and size of the fields in the mbuf structure changed, as described in the *New Features* section.

- The rte_cryptodev_info.sym structure has a new field max_nb_sessions_per_qp to support drivers which may support a limited number of sessions per queue_pair.

19.13.6 Removed Items

- KNI vhost support has been removed.
- The dpdk_qat sample application has been removed.

19.13.7 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
+ librte_bitratestats.so.1
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_cryptodev.so.2
  librte_distributor.so.1
+ librte_eal.so.4
  librte_ethdev.so.6
+ librte_eventdev.so.1
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
+ librte_latencystats.so.1
  librte_lpm.so.2
+ librte_mbuf.so.3
  librte_mempool.so.2
  librte_meter.so.1
+ librte_metrics.so.1
  librte_net.so.1
  librte_pdump.so.1
  librte_pipeline.so.3
  librte_pmd_bond.so.1
  librte_pmd_ring.so.2
```

(continues on next page)

(continued from previous page)

```
librte_port.so.3
librte_power.so.1
librte_reorder.so.1
librte_ring.so.1
librte_sched.so.1
librte_table.so.2
librte_timer.so.1
librte_vhost.so.3
```

19.13.8 Tested Platforms

- Intel(R) platforms with Intel(R) NICs combinations
  - CPU
    - Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
    - Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
    - Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
    - Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-2658 v2 @ 2.40GHz
    - Intel(R) Xeon(R) CPU E5-2658 v3 @ 2.20GHz
  - OS:
    - CentOS 7.2
    - Fedora 25
    - FreeBSD 11
    - Red Hat Enterprise Linux Server release 7.3
    - SUSE Enterprise Linux 12
    - Wind River Linux 8
    - Ubuntu 16.04
    - Ubuntu 16.10
  - NICs:
    - Intel(R) 82599ES 10 Gigabit Ethernet Controller
      - Firmware version: 0x61bf0001
      - Device id (pf/vf): 8086:10fb / 8086:10ed
      - Driver version: 4.0.1-k (ixgbe)
    - Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      - Firmware version: 0x800001cf

            - Device id (pf/vf): 8086:15ad / 8086:15a8
            - Driver version: 4.2.5 (ixgbe)
        * Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
            - Firmware version: 5.05
            - Device id (pf/vf): 8086:1572 / 8086:154c
            - Driver version: 1.5.23 (i40e)
        * Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
            - Firmware version: 5.05
            - Device id (pf/vf): 8086:1572 / 8086:154c
            - Driver version: 1.5.23 (i40e)
        * Intel(R) Ethernet Converged Network Adapter XL710-QDA1 (1x40G)
            - Firmware version: 5.05
            - Device id (pf/vf): 8086:1584 / 8086:154c
            - Driver version: 1.5.23 (i40e)
        * Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
            - Firmware version: 5.05
            - Device id (pf/vf): 8086:1583 / 8086:154c
            - Driver version: 1.5.23 (i40e)
        * Intel(R) Corporation I350 Gigabit Network Connection
            - Firmware version: 1.48, 0x800006e7
            - Device id (pf/vf): 8086:1521 / 8086:1520
            - Driver version: 5.2.13-k (igb)
- Intel(R) platforms with Mellanox(R) NICs combinations
    - Platform details:
        * Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
        * Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
        * Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
        * Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
    - OS:
        * Red Hat Enterprise Linux Server release 7.3 (Maipo)
        * Red Hat Enterprise Linux Server release 7.2 (Maipo)
        * Ubuntu 16.10
        * Ubuntu 16.04
        * Ubuntu 14.04

- MLNX_OFED: 4.0-2.0.0.0
- NICs:
  - Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1007
    - Firmware version: 2.40.5030
  - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.2000
  - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.2000
  - Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.2000
  - Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.2000
  - Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.2000
  - Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - Firmware version: 12.18.2000
  - Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - Firmware version: 12.18.2000

    * Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000
    * Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000
    * Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000
    * Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000
    * Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1015
        - Firmware version: 14.18.2000
    * Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1015
        - Firmware version: 14.18.2000
    * Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1017
        - Firmware version: 16.19.1200
    * Mellanox(R) ConnectX-5 Ex EN 100G MCX516A-CDAT (2x100G)
        - Host interface: PCI Express 4.0 x16
        - Device ID: 15b3:1019
        - Firmware version: 16.19.1200

- IBM(R) Power8(R) with Mellanox(R) NICs combinations
  - Platform details:

    - Processor: POWER8E (raw), AltiVec supported
    - type-model: 8247-22L
    - Firmware FW810.21 (SV810_108)
- OS: Ubuntu 16.04 LTS PPC le
- MLNX_OFED: 4.0-2.0.0.0
- NICs:
    - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000
    - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000
    - Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000
    - Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000
    - Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000
    - Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
        - Host interface: PCI Express 3.0 x16
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000
    - Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
        - Host interface: PCI Express 3.0 x8
        - Device ID: 15b3:1013
        - Firmware version: 12.18.2000

* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000
* Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.2000

19.14 DPDK Release 17.02

19.14.1 New Features

- **Added support for representing buses in EAL**

  The `rte_bus` structure was introduced into the EAL. This allows for devices to be represented by buses they are connected to. A new bus can be added to DPDK by extending the `rte_bus` structure and implementing the scan and probe functions. Once a new bus is registered using the provided APIs, new devices can be detected and initialized using bus scan and probe callbacks.

  With this change, devices other than PCI or VDEV type can be represented in the DPDK framework.

- **Added generic EAL API for I/O device memory read/write operations.**

  This API introduces 8 bit, 16 bit, 32 bit and 64 bit I/O device memory read/write operations along with "relaxed" versions.

  Weakly-ordered architectures like ARM need an additional I/O barrier for device memory read/write access over PCI bus. By introducing the EAL abstraction for I/O device memory read/write access, the drivers can access I/O device memory in an architecture-agnostic manner. The relaxed version does not have an additional I/O memory barrier, which is useful in accessing the device registers of integrated controllers which is implicitly strongly ordered with respect to memory access.

- **Added generic flow API (rte_flow).**

  This API provides a generic means to configure hardware to match specific ingress or egress traffic, alter its behavior and query related counters according to any number of user-defined rules.

  In order to expose a single interface with an unambiguous behavior that is common to all poll-mode drivers (PMDs) the `rte_flow` API is slightly higher-level than the legacy filtering framework, which it encompasses and supersedes (including all functions and filter types) .

  See the *Generic flow API (rte_flow)* documentation for more information.

- **Added firmware version get API.**

  Added a new function `rte_eth_dev_fw_version_get()` to fetch the firmware version for a given device.

- **Added APIs for MACsec offload support to the ixgbe PMD.**

  Six new APIs have been added to the ixgbe PMD for MACsec offload support. The declarations for the APIs can be found in `rte_pmd_ixgbe.h`.

- **Added I219 NICs support.**

  Added support for I219 Intel 1GbE NICs.

- **Added VF Daemon (VFD) for i40e. - EXPERIMENTAL**

  This is an EXPERIMENTAL feature to enhance the capability of the DPDK PF as many VF management features are not currently supported by the kernel PF driver. Some new private APIs are implemented directly in the PMD without an abstraction layer. They can be used directly by some users who have the need.

  The new APIs to control VFs directly from PF include:

  - Set VF MAC anti-spoofing.
  - Set VF VLAN anti-spoofing.
  - Set TX loopback.
  - Set VF unicast promiscuous mode.
  - Set VF multicast promiscuous mode.
  - Set VF MTU.
  - Get/reset VF stats.
  - Set VF MAC address.
  - Set VF VLAN stripping.
  - Vf VLAN insertion.
  - Set VF broadcast mode.
  - Set VF VLAN tag.
  - Set VF VLAN filter.

  VFD also includes VF to PF mailbox message management from an application. When the PF receives mailbox messages from the VF the PF should call the callback provided by the application to know if they're permitted to be processed.

As an EXPERIMENTAL feature, please be aware it can be changed or even removed without prior notice.

- **Updated the i40e base driver.**

  Updated the i40e base driver, including the following changes:

  - Replace existing legacy `memcpy()` calls with `i40e_memcpy()` calls.
  - Use `BIT()` macro instead of bit fields.
  - Add clear all WoL filters implementation.
  - Add broadcast promiscuous control per VLAN.
  - Remove unused `X722_SUPPORT` and `I40E_NDIS_SUPPORT` macros.

- **Updated the enic driver.**

  - Set new Rx checksum flags in mbufs to indicate unknown, good or bad checksums.
  - Fix set/remove of MAC addresses. Allow up to 64 addresses per device.
  - Enable TSO on outer headers.

- **Added Solarflare libefx-based network PMD.**

  Added a new network PMD which supports Solarflare SFN7xxx and SFN8xxx family of 10/40 Gbps adapters.

- **Updated the mlx4 driver.**

  - Addressed a few bugs.

- **Added support for Mellanox ConnectX-5 adapters (mlx5).**

  Added support for Mellanox ConnectX-5 family of 10/25/40/50/100 Gbps adapters to the existing mlx5 PMD.

- **Updated the mlx5 driver.**

  - Improve Tx performance by using vector logic.
  - Improve RSS balancing when number of queues is not a power of two.
  - Generic flow API support for Ethernet, IPv4, IPv4, UDP, TCP, VLAN and VXLAN pattern items with DROP and QUEUE actions.
  - Support for extended statistics.
  - Addressed several data path bugs.
  - As of MLNX_OFED 4.0-1.0.1.0, the Toeplitz RSS hash function is not symmetric anymore for consistency with other PMDs.

- **virtio-user with vhost-kernel as another exceptional path.**

  Previously, we upstreamed a virtual device, virtio-user with vhost-user as the backend as a way of enabling IPC (Inter-Process Communication) and user space container networking.

  Virtio-user with vhost-kernel as the backend is a solution for the exception path, such as KNI, which exchanges packets with the kernel networking stack. This solution is very promising in:

  - Maintenance: vhost and vhost-net (kernel) is an upstreamed and extensively used kernel module.

  - Features: vhost-net is designed to be a networking solution, which has lots of networking related features, like multi-queue, TSO, multi-seg mbuf, etc.
  - Performance: similar to KNI, this solution would use one or more kthreads to send/receive packets from user space DPDK applications, which has little impact on user space polling thread (except that it might enter into kernel space to wake up those kthreads if necessary).

- **Added virtio Rx interrupt support.**

  Added a feature to enable Rx interrupt mode for virtio pci net devices as bound to VFIO (noiommu mode) and driven by virtio PMD.

  With this feature, the virtio PMD can switch between polling mode and interrupt mode, to achieve best performance, and at the same time save power. It can work on both legacy and modern virtio devices. In this mode, each `rxq` is mapped with an excluded MSIx interrupt.

  See the *Virtio Interrupt Mode* documentation for more information.

- **Added ARMv8 crypto PMD.**

  A new crypto PMD has been added, which provides combined mode cryptographic operations optimized for ARMv8 processors. The driver can be used to enhance performance in processing chained operations such as cipher + HMAC.

- **Updated the QAT PMD.**

  The QAT PMD has been updated with additional support for:

  - DES algorithm.
  - Scatter-gather list (SGL) support.

- **Updated the AESNI MB PMD.**

  - The Intel(R) Multi Buffer Crypto for IPsec library used in AESNI MB PMD has been moved to a new repository, in GitHub.
  - Support has been added for single operations (cipher only and authentication only).

- **Updated the AES-NI GCM PMD.**

  The AES-NI GCM PMD was migrated from the Multi Buffer library to the ISA-L library. The migration entailed adding additional support for:

  - GMAC algorithm.
  - 256-bit cipher key.
  - Session-less mode.
  - Out-of place processing
  - Scatter-gather support for chained mbufs (only out-of place and destination mbuf must be contiguous)

- **Added crypto performance test application.**

  Added a new performance test application for measuring performance parameters of PMDs available in the crypto tree.

- **Added Elastic Flow Distributor library (rte_efd).**

  Added a new library which uses perfect hashing to determine a target/value for a given incoming flow key.

The library does not store the key itself for lookup operations, and therefore, lookup performance is not dependent on the key size. Also, the target/value can be any arbitrary value (8 bits by default). Finally, the storage requirement is much smaller than a hash-based flow table and therefore, it can better fit in CPU cache and scale to millions of flow keys.

See the *Elastic Flow Distributor Library* documentation in the Programmers Guide document, for more information.

19.14.2 Resolved Issues

Drivers

- **net/virtio: Fixed multiple process support.**

  Fixed a few regressions introduced in recent releases that break the virtio multiple process support.

Examples

- **examples/ethtool: Fixed crash with non-PCI devices.**

  Fixed issue where querying a non-PCI device was dereferencing non-existent PCI data resulting in a segmentation fault.

19.14.3 API Changes

- **Moved five APIs for VF management from the ethdev to the ixgbe PMD.**

  The following five APIs for VF management from the PF have been removed from the ethdev, renamed, and added to the ixgbe PMD:

```
rte_eth_dev_set_vf_rate_limit()
rte_eth_dev_set_vf_rx()
rte_eth_dev_set_vf_rxmode()
rte_eth_dev_set_vf_tx()
rte_eth_dev_set_vf_vlan_filter()
```

  The API's have been renamed to the following:

```
rte_pmd_ixgbe_set_vf_rate_limit()
rte_pmd_ixgbe_set_vf_rx()
rte_pmd_ixgbe_set_vf_rxmode()
rte_pmd_ixgbe_set_vf_tx()
rte_pmd_ixgbe_set_vf_vlan_filter()
```

  The declarations for the API's can be found in `rte_pmd_ixgbe.h`.

19.14.4 ABI Changes

19.14.5 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_cryptodev.so.2
  librte_distributor.so.1
  librte_eal.so.3
+ librte_ethdev.so.6
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_lpm.so.2
  librte_mbuf.so.2
  librte_mempool.so.2
  librte_meter.so.1
  librte_net.so.1
  librte_pdump.so.1
  librte_pipeline.so.3
  librte_pmd_bond.so.1
  librte_pmd_ring.so.2
  librte_port.so.3
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
  librte_table.so.2
  librte_timer.so.1
  librte_vhost.so.3
```

19.14.6 Tested Platforms

This release has been tested with the below list of CPU/device/firmware/OS. Each section describes a different set of combinations.

- Intel(R) platforms with Mellanox(R) NICs combinations
  - Platform details
    - Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
  - OS:
    - CentOS 7.0
    - Fedora 23
    - Fedora 24
    - FreeBSD 10.3

    - Red Hat Enterprise Linux 7.2
    - SUSE Enterprise Linux 12
    - Ubuntu 14.04 LTS
    - Ubuntu 15.10
    - Ubuntu 16.04 LTS
    - Wind River Linux 8
  - MLNX_OFED: 4.0-1.0.1.0
  - NICs:
    - Mellanox(R) ConnectX(R)-3 Pro 40G MCX354A-FCC_Ax (2x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1007
      - Firmware version: 2.40.5030
    - Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
    - Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
    - Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
    - Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
    - Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
      - Host interface: PCI Express 3.0 x8
      - Device ID: 15b3:1013
      - Firmware version: 12.18.1000
    - Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
      - Host interface: PCI Express 3.0 x16

- Device ID: 15b3:1013
- Firmware version: 12.18.1000

* Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.18.1000
* Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.18.1000
* Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1017
  - Firmware version: 16.18.1000

        * Mellanox(R) ConnectX-5 Ex EN 100G MCX516A-CDAT (2x100G)
            - Host interface: PCI Express 4.0 x16
            - Device ID: 15b3:1019
            - Firmware version: 16.18.1000
- IBM(R) Power8(R) with Mellanox(R) NICs combinations
    - Machine:
        * Processor: POWER8E (raw), AltiVec supported
            - type-model: 8247-22L
            - Firmware FW810.21 (SV810_108)
    - OS: Ubuntu 16.04 LTS PPC le
    - MLNX_OFED: 4.0-1.0.1.0
    - NICs:
        * Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1013
            - Firmware version: 12.18.1000
        * Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1013
            - Firmware version: 12.18.1000
        * Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1013
            - Firmware version: 12.18.1000
        * Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1013
            - Firmware version: 12.18.1000
        * Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
            - Host interface: PCI Express 3.0 x8
            - Device ID: 15b3:1013
            - Firmware version: 12.18.1000
        * Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
            - Host interface: PCI Express 3.0 x16

- Device ID: 15b3:1013
- Firmware version: 12.18.1000

* Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1013
  - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1013
  - Firmware version: 12.18.1000
* Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.18.1000
* Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
  - Host interface: PCI Express 3.0 x8
  - Device ID: 15b3:1015
  - Firmware version: 14.18.1000
* Mellanox(R) ConnectX(R)-5 100G MCX556A-ECAT (2x100G)
  - Host interface: PCI Express 3.0 x16
  - Device ID: 15b3:1017
  - Firmware version: 16.18.1000

- Intel(R) platforms with Intel(R) NICs combinations
  - Platform details
    - Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
    - Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
    - Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
    - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
    - Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
    - Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
    - Intel(R) Xeon(R) CPU E5-2658 v2 @ 2.40GHz
  - OS:
    - CentOS 7.2
    - Fedora 25
    - FreeBSD 11
    - Red Hat Enterprise Linux Server release 7.3
    - SUSE Enterprise Linux 12
    - Wind River Linux 8
    - Ubuntu 16.04
    - Ubuntu 16.10
  - NICs:
    - Intel(R) 82599ES 10 Gigabit Ethernet Controller
      - Firmware version: 0x61bf0001
      - Device id (pf/vf): 8086:10fb / 8086:10ed
      - Driver version: 4.0.1-k (ixgbe)
    - Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
      - Firmware version: 0x800001cf
      - Device id (pf/vf): 8086:15ad / 8086:15a8
      - Driver version: 4.2.5 (ixgbe)
    - Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
      - Firmware version: 5.05
      - Device id (pf/vf): 8086:1572 / 8086:154c
      - Driver version: 1.5.23 (i40e)
    - Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
      - Firmware version: 5.05
      - Device id (pf/vf): 8086:1572 / 8086:154c

        - Driver version: 1.5.23 (i40e)
    * Intel(R) Ethernet Converged Network Adapter XL710-QDA1 (1x40G)
        - Firmware version: 5.05
        - Device id (pf/vf): 8086:1584 / 8086:154c
        - Driver version: 1.5.23 (i40e)
    * Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
        - Firmware version: 5.05
        - Device id (pf/vf): 8086:1583 / 8086:154c
        - Driver version: 1.5.23 (i40e)
    * Intel(R) Corporation I350 Gigabit Network Connection
        - Firmware version: 1.48, 0x800006e7
        - Device id (pf/vf): 8086:1521 / 8086:1520
        - Driver version: 5.2.13-k (igb)

19.15 DPDK Release 16.11

19.15.1 New Features

- **Added software parser for packet type.**
  - Added a new function `rte_pktmbuf_read()` to read the packet data from an mbuf chain, linearizing if required.
  - Added a new function `rte_net_get_ptype()` to parse an Ethernet packet in an mbuf chain and retrieve its packet type from software.
  - Added new functions `rte_get_ptype_*()` to dump a packet type as a string.
- **Improved offloads support in mbuf.**
  - Added a new function `rte_raw_cksum_mbuf()` to process the checksum of data embedded in an mbuf chain.
  - Added new Rx checksum flags in mbufs to describe more states: unknown, good, bad, or not present (useful for virtual drivers). This modification was done for IP and L4.
  - Added a new Rx LRO mbuf flag, used when packets are coalesced. This flag indicates that the segment size of original packets is known.
- **Added vhost-user dequeue zero copy support.**

  The copy in the dequeue path is avoided in order to improve the performance. In the VM2VM case, the boost is quite impressive. The bigger the packet size, the bigger performance boost you may get. However, for the VM2NIC case, there are some limitations, so the boost is not as impressive as the VM2VM case. It may even drop quite a bit for small packets.

  For that reason, this feature is disabled by default. It can be enabled when the `RTE_VHOST_USER_DEQUEUE_ZERO_COPY` flag is set. Check the VHost section of the Programming Guide for more information.

- **Added vhost-user indirect descriptors support.**

  If the indirect descriptor feature is enabled, each packet sent by the guest will take exactly one slot in the enqueue virtqueue. Without this feature, as in the current version, even 64 bytes packets take two slots with Virtio PMD on guest side.

  The main impact is better performance for 0% packet loss use-cases, as it behaves as if the virtqueue size was enlarged, so more packets can be buffered in the case of system perturbations. On the downside, small performance degradations were measured when running micro-benchmarks.

- **Added vhost PMD xstats.**

  Added extended statistics to vhost PMD from a per port perspective.

- **Supported offloads with virtio.**

  Added support for the following offloads in virtio:

  - Rx/Tx checksums.
  - LRO.
  - TSO.

- **Added virtio NEON support for ARM.**

  Added NEON support for ARM based virtio.

- **Updated the ixgbe base driver.**

  Updated the ixgbe base driver, including the following changes:

  - Added X550em_a 10G PHY support.
  - Added support for flow control auto negotiation for X550em_a 1G PHY.
  - Added X550em_a FW ALEF support.
  - Increased mailbox version to `ixgbe_mbox_api_13`.
  - Added two MAC operations for Hyper-V support.

- **Added APIs for VF management to the ixgbe PMD.**

  Eight new APIs have been added to the ixgbe PMD for VF management from the PF. The declarations for the API's can be found in `rte_pmd_ixgbe.h`.

- **Updated the enic driver.**

  - Added update to use interrupt for link status checking instead of polling.
  - Added more flow director modes on UCS Blade with firmware version >= 2.0(13e).
  - Added full support for MTU update.
  - Added support for the `rte_eth_rx_queue_count` function.

- **Updated the mlx5 driver.**

  - Added support for RSS hash results.
  - Added several performance improvements.
  - Added several bug fixes.

- **Updated the QAT PMD.**

  The QAT PMD was updated with additional support for:

  - MD5_HMAC algorithm.
  - SHA224-HMAC algorithm.
  - SHA384-HMAC algorithm.
  - GMAC algorithm.
  - KASUMI (F8 and F9) algorithm.
  - 3DES algorithm.
  - NULL algorithm.
  - C3XXX device.
  - C62XX device.

- **Added openssl PMD.**

  A new crypto PMD has been added, which provides several ciphering and hashing algorithms. All cryptography operations use the Openssl library crypto API.

- **Updated the IPsec example.**

  Updated the IPsec example with the following support:

  - Configuration file support.
  - AES CBC IV generation with cipher forward function.
  - AES GCM/CTR mode.

- **Added support for new gcc -march option.**

  The GCC 4.9 `-march` option supports the Intel processor code names. The config option `RTE_MACHINE` can be used to pass code names to the compiler via the `-march` flag.

19.15.2 Resolved Issues

Drivers

- **enic: Fixed several flow director issues.**
- **enic: Fixed inadvertent setting of L4 checksum ptype on ICMP packets.**
- **enic: Fixed high driver overhead when servicing Rx queues beyond the first.**

19.15.3 Known Issues

- **L3fwd-power app does not work properly when Rx vector is enabled.**

  The L3fwd-power app doesn't work properly with some drivers in vector mode since the queue monitoring works differently between scalar and vector modes leading to incorrect frequency scaling. In addition, L3fwd-power application requires the mbuf to have correct packet type set but in some drivers the vector mode must be disabled for this.

  Therefore, in order to use L3fwd-power, vector mode should be disabled via the config file.

- **Digest address must be supplied for crypto auth operation on QAT PMD.**

  The cryptodev API specifies that if the rte_crypto_sym_op.digest.data field, and by inference the digest.phys_addr field which points to the same location, is not set for an auth operation the driver is to understand that the digest result is located immediately following the region over which the digest is computed. The QAT PMD doesn't correctly handle this case and reads and writes to an incorrect location.

  Callers can workaround this by always supplying the digest virtual and physical address fields in the rte_crypto_sym_op for an auth operation.

19.15.4 API Changes

- The driver naming convention has been changed to make them more consistent. It especially impacts `--vdev` arguments. For example `eth_pcap` becomes `net_pcap` and `cryptodev_aesni_mb_pmd` becomes `crypto_aesni_mb`.

  For backward compatibility an alias feature has been enabled to support the original names.

- The log history has been removed.

- The `rte_ivshmem` feature (including library and EAL code) has been removed in 16.11 because it had some design issues which were not planned to be fixed.

- The `file_name` data type of `struct rte_port_source_params` and `struct rte_port_sink_params` is changed from `char *` to `const char *`.

- **Improved device/driver hierarchy and generalized hotplugging.**

  The device and driver relationship has been restructured by introducing generic classes. This paves the way for having PCI, VDEV and other device types as instantiated objects rather than classes in themselves. Hotplugging has also been generalized into EAL so that Ethernet or crypto devices can use the common infrastructure.

  - Removed `pmd_type` as a way of segregation of devices.
  - Moved `numa_node` and `devargs` into `rte_driver` from `rte_pci_driver`. These can now be used by any instantiated object of `rte_driver`.
  - Added `rte_device` class and all PCI and VDEV devices inherit from it
  - Renamed devinit/devuninit handlers to probe/remove to make it more semantically correct with respect to the device <=> driver relationship.
  - Moved hotplugging support to EAL. Hereafter, PCI and vdev can use the APIs `rte_eal_dev_attach` and `rte_eal_dev_detach`.

- Renamed helpers and support macros to make them more synonymous with their device types (e.g. `PMD_REGISTER_DRIVER` => `RTE_PMD_REGISTER_PCI`).
- Device naming functions have been generalized from ethdev and cryptodev to EAL. `rte_eal_pci_device_name` has been introduced for obtaining unique device name from PCI Domain-BDF description.
- Virtual device registration APIs have been added: `rte_eal_vdrv_register` and `rte_eal_vdrv_unregister`.

19.15.5 ABI Changes

19.15.6 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
  librte_acl.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
+ librte_cryptodev.so.2
  librte_distributor.so.1
+ librte_eal.so.3
+ librte_ethdev.so.5
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_lpm.so.2
  librte_mbuf.so.2
  librte_mempool.so.2
  librte_meter.so.1
  librte_net.so.1
  librte_pdump.so.1
  librte_pipeline.so.3
  librte_pmd_bond.so.1
  librte_pmd_ring.so.2
  librte_port.so.3
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
  librte_table.so.2
  librte_timer.so.1
  librte_vhost.so.3
```

19.15.7 Tested Platforms

1. SuperMicro 1U
   - BIOS: 1.0c
   - Processor: Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
2. SuperMicro 1U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz

- Onboard NIC: Intel(R) X552/X557-AT (2x10G)
  - Firmware-version: 0x800001cf
  - Device ID (PF/VF): 8086:15ad /8086:15a8
- kernel driver version: 4.2.5 (ixgbe)

3. SuperMicro 2U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
4. Intel(R) Server board S2600GZ
   - BIOS: SE5C600.86B.02.02.0002.122320131210
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
5. Intel(R) Server board W2600CR
   - BIOS: SE5C600.86B.02.01.0002.082220131453
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
6. Intel(R) Server board S2600CWT
   - BIOS: SE5C610.86B.01.01.0009.060120151350
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
7. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.01.01.0005.101720141054
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
8. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.11.01.0044.090120151156
   - Processor: Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
9. Intel(R) Server board S2600WTT
   - Processor: Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz
10. Intel(R) Server
    - Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
11. IBM(R) Power8(R)
    - Machine type-model: 8247-22L
    - Firmware FW810.21 (SV810_108)
    - Processor: POWER8E (raw), AltiVec supported

19.15.8 Tested NICs

1. Intel(R) Ethernet Controller X540-AT2
   - Firmware version: 0x80000389
   - Device id (pf): 8086:1528
   - Driver version: 3.23.2 (ixgbe)
2. Intel(R) 82599ES 10 Gigabit Ethernet Controller
   - Firmware version: 0x61bf0001
   - Device id (pf/vf): 8086:10fb / 8086:10ed
   - Driver version: 4.0.1-k (ixgbe)
3. Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
   - Firmware version: 0x800001cf
   - Device id (pf/vf): 8086:15ad / 8086:15a8
   - Driver version: 4.2.5 (ixgbe)
4. Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
   - Firmware version: 5.05
   - Device id (pf/vf): 8086:1572 / 8086:154c
   - Driver version: 1.5.23 (i40e)
5. Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
   - Firmware version: 5.05
   - Device id (pf/vf): 8086:1572 / 8086:154c
   - Driver version: 1.5.23 (i40e)
6. Intel(R) Ethernet Converged Network Adapter XL710-QDA1 (1x40G)
   - Firmware version: 5.05
   - Device id (pf/vf): 8086:1584 / 8086:154c
   - Driver version: 1.5.23 (i40e)
7. Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
   - Firmware version: 5.05
   - Device id (pf/vf): 8086:1583 / 8086:154c
   - Driver version: 1.5.23 (i40e)
8. Intel(R) Corporation I350 Gigabit Network Connection
   - Firmware version: 1.48, 0x800006e7
   - Device id (pf/vf): 8086:1521 / 8086:1520
   - Driver version: 5.2.13-k (igb)
9. Intel(R) Ethernet Multi-host Controller FM10000

- Firmware version: N/A
- Device id (pf/vf): 8086:15d0
- Driver version: 0.17.0.9 (fm10k)

10. Mellanox(R) ConnectX(R)-4 10G MCX4111A-XCAT (1x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
11. Mellanox(R) ConnectX(R)-4 10G MCX4121A-XCAT (2x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
12. Mellanox(R) ConnectX(R)-4 25G MCX4111A-ACAT (1x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
13. Mellanox(R) ConnectX(R)-4 25G MCX4121A-ACAT (2x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
14. Mellanox(R) ConnectX(R)-4 40G MCX4131A-BCAT/MCX413A-BCAT (1x40G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
15. Mellanox(R) ConnectX(R)-4 40G MCX415A-BCAT (1x40G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
16. Mellanox(R) ConnectX(R)-4 50G MCX4131A-GCAT/MCX413A-GCAT (1x50G)

- Host interface: PCI Express 3.0 x8
- Device ID: 15b3:1013
- MLNX_OFED: 3.4-1.0.0.0
- Firmware version: 12.17.1010

17. Mellanox(R) ConnectX(R)-4 50G MCX414A-BCAT (2x50G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
18. Mellanox(R) ConnectX(R)-4 50G MCX415A-GCAT/MCX416A-BCAT/MCX416A-GCAT (2x50G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
19. Mellanox(R) ConnectX(R)-4 50G MCX415A-CCAT (1x100G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
20. Mellanox(R) ConnectX(R)-4 100G MCX416A-CCAT (2x100G)
    - Host interface: PCI Express 3.0 x16
    - Device ID: 15b3:1013
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 12.17.1010
21. Mellanox(R) ConnectX(R)-4 Lx 10G MCX4121A-XCAT (2x10G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1015
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 14.17.1010
22. Mellanox(R) ConnectX(R)-4 Lx 25G MCX4121A-ACAT (2x25G)
    - Host interface: PCI Express 3.0 x8
    - Device ID: 15b3:1015
    - MLNX_OFED: 3.4-1.0.0.0
    - Firmware version: 14.17.1010

19.15.9 Tested OSes

- CentOS 7.2
- Fedora 23
- Fedora 24
- FreeBSD 10.3
- FreeBSD 11
- Red Hat Enterprise Linux Server release 6.7 (Santiago)
- Red Hat Enterprise Linux Server release 7.0 (Maipo)
- Red Hat Enterprise Linux Server release 7.2 (Maipo)
- SUSE Enterprise Linux 12
- Wind River Linux 6.0.0.26
- Wind River Linux 8
- Ubuntu 14.04
- Ubuntu 15.04
- Ubuntu 16.04

19.16 DPDK Release 16.07

19.16.1 New Features

- **Removed the mempool cache memory if caching is not being used.**

  The size of the mempool structure is reduced if the per-lcore cache is disabled.

- **Added mempool external cache for non-EAL thread.**

  Added new functions to create, free or flush a user-owned mempool cache for non-EAL threads. Previously the caching was always disabled on these threads.

- **Changed the memory allocation scheme in the mempool library.**
  - Added the ability to allocate a large mempool in fragmented virtual memory.
  - Added new APIs to populate a mempool with memory.
  - Added an API to free a mempool.
  - Modified the API of the `rte_mempool_obj_iter()` function.
  - Dropped the specific Xen Dom0 code.
  - Dropped the specific anonymous mempool code in testpmd.

- **Added a new driver for Broadcom NetXtreme-C devices.**

  Added the new bnxt driver for Broadcom NetXtreme-C devices. See the "Network Interface Controller Drivers" document for more details on this new driver.

- **Added a new driver for ThunderX nicvf devices.**

  Added the new thunderx net driver for ThunderX nicvf devices. See the "Network Interface Controller Drivers" document for more details on this new driver.

- **Added mailbox interrupt support for ixgbe and igb VFs.**

  When the physical NIC link comes up or down, the PF driver will send a mailbox message to notify each VF. To handle this link up/down event, support have been added for a mailbox interrupt to receive the message and allow the application to register a callback for it.

- **Updated the ixgbe base driver.**

  The ixgbe base driver was updated with changes including the following:

  - Added sgmii link for X550.
  - Added MAC link setup for X550a SFP and SFP+.
  - Added KR support for X550em_a.
  - Added new PHY definitions for M88E1500.
  - Added support for the VLVF to be bypassed when adding/removing a VFTA entry.
  - Added X550a flow control auto negotiation support.

- **Updated the i40e base driver.**

  Updated the i40e base driver including support for new devices IDs.

- **Updated the enic driver.**

  The enic driver was updated with changes including the following:

  - Optimized the Tx function.
  - Added Scattered Rx capability.
  - Improved packet type identification.
  - Added MTU update in non Scattered Rx mode and enabled MTU of up to 9208 with UCS Software release 2.2 on 1300 series VICs.

- **Updated the mlx5 driver.**

  The mlx5 driver was updated with changes including the following:

  - Data path was refactored to bypass Verbs to improve RX and TX performance.
  - Removed compilation parameters for inline send, `MLX5_MAX_INLINE`, and added command line parameter instead, `txq_inline`.
  - Improved TX scatter gather support: Removed compilation parameter `MLX5_PMD_SGE_WR_N`. Scatter-gather elements is set to the maximum value the NIC supports. Removed linearization logic, this decreases the memory consumption of the PMD.
  - Improved jumbo frames support, by dynamically setting RX scatter gather elements according to the MTU and mbuf size, no need for compilation parameter `MLX5_PMD_SGE_WR_N`

- **Added support for virtio on IBM POWER8.**

  The ioports are mapped in memory when using Linux UIO.

- **Added support for Virtio in containers.**

  Add a new virtual device, named virtio_user, to support virtio for containers.

  Known limitations:

  - Control queue and multi-queue are not supported yet.
  - Doesn't work with `--huge-unlink`.
  - Doesn't work with `--no-huge`.
  - Doesn't work when there are more than `VHOST_MEMORY_MAX_NREGIONS(8)` hugepages.
  - Root privilege is required for sorting hugepages by physical address.
  - Can only be used with the vhost user backend.

- **Added vhost-user client mode.**

  DPDK vhost-user now supports client mode as well as server mode. Client mode is enabled when the `RTE_VHOST_USER_CLIENT` flag is set while calling `rte_vhost_driver_register`.

  When DPDK vhost-user restarts from an normal or abnormal exit (such as a crash), the client mode allows DPDK to establish the connection again. Note that QEMU version v2.7 or above is required for this feature.

  DPDK vhost-user will also try to reconnect by default when:

  - The first connect fails (for example when QEMU is not started yet).
  - The connection is broken (for example when QEMU restarts).

  It can be turned off by setting the `RTE_VHOST_USER_NO_RECONNECT` flag.

- **Added NSH packet recognition in i40e.**

- **Added AES-CTR support to AESNI MB PMD.**

  Now AESNI MB PMD supports 128/192/256-bit counter mode AES encryption and decryption.

- **Added AES counter mode support for Intel QuickAssist devices.**

  Enabled support for the AES CTR algorithm for Intel QuickAssist devices. Provided support for algorithm-chaining operations.

- **Added KASUMI SW PMD.**

  A new Crypto PMD has been added, which provides KASUMI F8 (UEA1) ciphering and KASUMI F9 (UIA1) hashing.

- **Added multi-writer support for RTE Hash with Intel TSX.**

  The following features/modifications have been added to rte_hash library:

  - Enabled application developers to use an extra flag for `rte_hash` creation to specify default behavior (multi-thread safe/unsafe) with the `rte_hash_add_key` function.
  - Changed the Cuckoo Hash Search algorithm to breadth first search for multi-writer routines and split Cuckoo Hash Search and Move operations in order to reduce transactional code region and improve TSX performance.
  - Added a hash multi-writer test case to the test app.

- **Improved IP Pipeline Application.**

  The following features have been added to the ip_pipeline application:

  - Configure the MAC address in the routing pipeline and automatic route updates with change in link state.
  - Enable RSS per network interface through the configuration file.
  - Streamline the CLI code.

- **Added keepalive enhancements.**

  Added support for reporting of core states other than "dead" to monitoring applications, enabling the support of broader liveness reporting to external processes.

- **Added packet capture framework.**

  - A new library `librte_pdump` is added to provide a packet capture API.
  - A new `app/pdump` tool is added to demonstrate capture packets in DPDK.

- **Added floating VEB support for i40e PF driver.**

  A "floating VEB" is a special Virtual Ethernet Bridge (VEB) which does not have an upload port, but instead is used for switching traffic between virtual functions (VFs) on a port.

  For information on this feature, please see the "I40E Poll Mode Driver" section of the "Network Interface Controller Drivers" document.

- **Added support for live migration of a VM with SRIOV VF.**

  Live migration of a VM with Virtio and VF PMD's using the bonding PMD.

19.16.2 Resolved Issues

EAL

- **igb_uio: Fixed possible mmap failure for Linux >= 4.5.**

  The mmaping of the iomem range of the PCI device fails for kernels that enabled the `CONFIG_IO_STRICT_DEVMEM` option. The error seen by the user is as similar to the following:

```
EAL: pci_map_resource():

    cannot mmap(39, 0x7f1c51800000, 0x100000, 0x0):
    Invalid argument (0xffffffffffffffff)
```

  The `CONFIG_IO_STRICT_DEVMEM` kernel option was introduced in Linux v4.5.

  The issues was resolve by updating `igb_uio` to stop reserving PCI memory resources. From the kernel point of view the iomem region looks like idle and mmap works again. This matches the `uio_pci_generic` usage.

Drivers

- **i40e: Fixed vlan stripping from inner header.**

  Previously, for tunnel packets, such as VXLAN/NVGRE, the vlan tags of the inner header will be stripped without putting vlan info to descriptor. Now this issue is fixed by disabling vlan stripping from inner header.

- **i40e: Fixed the type issue of a single VLAN type.**

  Currently, if a single VLAN header is added in a packet, it's treated as inner VLAN. But generally, a single VLAN header is treated as the outer VLAN header. This issue is fixed by changing corresponding register for single VLAN.

- **enic: Fixed several issues when stopping then restarting ports and queues.**

  Fixed several crashes related to stopping then restarting ports and queues. Fixed possible crash when re-configuring the number of Rx queue descriptors.

- **enic: Fixed Rx data mis-alignment if mbuf data offset modified.**

  Fixed possible Rx corruption when mbufs were returned to a pool with data offset other than RTE_PKTMBUF_HEADROOM.

- **enic: Fixed Tx IP/UDP/TCP checksum offload and VLAN insertion.**

- **enic: Fixed Rx error and missed counters.**

Libraries

- **mbuf: Fixed refcnt update when detaching.**

  Fix the `rte_pktmbuf_detach()` function to decrement the direct mbuf's reference counter. The previous behavior was not to affect the reference counter. This lead to a memory leak of the direct mbuf.

Examples

Other

19.16.3 Known Issues

19.16.4 API Changes

- The following counters are removed from the `rte_eth_stats` structure:
  - `ibadcrc`
  - `ibadlen`
  - `imcasts`
  - `fdirmatch`
  - `fdirmiss`
  - `tx_pause_xon`

  - rx_pause_xon
  - tx_pause_xoff
  - rx_pause_xoff

- The extended statistics are fetched by ids with `rte_eth_xstats_get` after a lookup by name `rte_eth_xstats_get_names`.
- The function `rte_eth_dev_info_get` fill the new fields `nb_rx_queues` and `nb_tx_queues` in the structure `rte_eth_dev_info`.
- The vhost function `rte_vring_available_entries` is renamed to `rte_vhost_avail_entries`.
- All existing vhost APIs and callbacks with `virtio_net` struct pointer as the parameter have been changed due to the ABI refactoring described below. It is replaced by `int vid`.
- The function `rte_vhost_enqueue_burst` no longer supports concurrent enqueuing packets to the same queue.
- The function `rte_eth_dev_set_mtu` adds a new return value `-EBUSY`, which indicates the operation is forbidden because the port is running.
- The script `dpdk_nic_bind.py` is renamed to `dpdk-devbind.py`. And the script `setup.sh` is renamed to `dpdk-setup.sh`.

19.16.5 ABI Changes

- The `rte_port_source_params` structure has new fields to support PCAP files. It was already in release 16.04 with `RTE_NEXT_ABI` flag.
- The `rte_eth_dev_info` structure has new fields `nb_rx_queues` and `nb_tx_queues` to support the number of queues configured by software.
- A Vhost ABI refactoring has been made: the `virtio_net` structure is no longer exported directly to the application. Instead, a handle, `vid`, has been used to represent this structure internally.

19.16.6 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
+ libethdev.so.4
  librte_acl.so.2
  librte_cfgfile.so.2
  librte_cmdline.so.2
  librte_cryptodev.so.1
  librte_distributor.so.1
  librte_eal.so.2
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_ivshmem.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_lpm.so.2
  librte_mbuf.so.2
+ librte_mempool.so.2
```

(continues on next page)

(continued from previous page)

```
  librte_meter.so.1
  librte_pdump.so.1
  librte_pipeline.so.3
  librte_pmd_bond.so.1
  librte_pmd_ring.so.2
+ librte_port.so.3
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
  librte_table.so.2
  librte_timer.so.1
+ librte_vhost.so.3
```

19.16.7 Tested Platforms

1. SuperMicro 1U
   - BIOS: 1.0c
   - Processor: Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
2. SuperMicro 1U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
   - Onboard NIC: Intel(R) X552/X557-AT (2x10G)
     - Firmware-version: 0x800001cf
     - Device ID (PF/VF): 8086:15ad /8086:15a8
   - kernel driver version: 4.2.5 (ixgbe)
3. SuperMicro 2U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
4. Intel(R) Server board S2600GZ
   - BIOS: SE5C600.86B.02.02.0002.122320131210
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
5. Intel(R) Server board W2600CR
   - BIOS: SE5C600.86B.02.01.0002.082220131453
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
6. Intel(R) Server board S2600CWT
   - BIOS: SE5C610.86B.01.01.0009.060120151350
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
7. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.01.01.0005.101720141054

- Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz

8. Intel(R) Server board S2600WTT
    - BIOS: SE5C610.86B.11.01.0044.090120151156
    - Processor: Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz

19.16.8 Tested NICs

1. Intel(R) Ethernet Controller X540-AT2
    - Firmware version: 0x80000389
    - Device id (pf): 8086:1528
    - Driver version: 3.23.2 (ixgbe)
2. Intel(R) 82599ES 10 Gigabit Ethernet Controller
    - Firmware version: 0x61bf0001
    - Device id (pf/vf): 8086:10fb / 8086:10ed
    - Driver version: 4.0.1-k (ixgbe)
3. Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
    - Firmware version: 0x800001cf
    - Device id (pf/vf): 8086:15ad / 8086:15a8
    - Driver version: 4.2.5 (ixgbe)
4. Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
    - Firmware version: 5.04
    - Device id (pf/vf): 8086:1572 / 8086:154c
    - Driver version: 1.4.26 (i40e)
5. Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
    - Firmware version: 5.04
    - Device id (pf/vf): 8086:1572 / 8086:154c
    - Driver version: 1.4.25 (i40e)
6. Intel(R) Ethernet Converged Network Adapter XL710-QDA1 (1x40G)
    - Firmware version: 5.04
    - Device id (pf/vf): 8086:1584 / 8086:154c
    - Driver version: 1.4.25 (i40e)
7. Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
    - Firmware version: 5.04
    - Device id (pf/vf): 8086:1583 / 8086:154c
    - Driver version: 1.4.25 (i40e)

8. Intel(R) Corporation I350 Gigabit Network Connection
    - Firmware version: 1.48, 0x800006e7
    - Device id (pf/vf): 8086:1521 / 8086:1520
    - Driver version: 5.2.13-k (igb)
9. Intel(R) Ethernet Multi-host Controller FM10000
    - Firmware version: N/A
    - Device id (pf/vf): 8086:15d0
    - Driver version: 0.17.0.9 (fm10k)

19.16.9 Tested OSes

- CentOS 7.0
- Fedora 23
- Fedora 24
- FreeBSD 10.3
- Red Hat Enterprise Linux 7.2
- SUSE Enterprise Linux 12
- Ubuntu 15.10
- Ubuntu 16.04 LTS
- Wind River Linux 8

19.17 DPDK Release 16.04

19.17.1 New Features

- **Added function to check primary process state.**

  A new function `rte_eal_primary_proc_alive()` has been added to allow the user to detect if a primary process is running. Use cases for this feature include fault detection, and monitoring using secondary processes.

- **Enabled bulk allocation of mbufs.**

  A new function `rte_pktmbuf_alloc_bulk()` has been added to allow the user to bulk allocate mbufs.

- **Added device link speed capabilities.**

  The structure `rte_eth_dev_info` now has a `speed_capa` bitmap, which allows the application to determine the supported speeds of each device.

- **Added bitmap of link speeds to advertise.**

  Added a feature to allow the definition of a set of advertised speeds for auto-negotiation, explicitly disabling link auto-negotiation (single speed) and full auto-negotiation.

- **Added new poll-mode driver for Amazon Elastic Network Adapters (ENA).**

  The driver operates for a variety of ENA adapters through feature negotiation with the adapter and upgradable commands set. The ENA driver handles PCI Physical and Virtual ENA functions.

- **Restored vmxnet3 TX data ring.**

  TX data ring has been shown to improve small packet forwarding performance on the vSphere environment.

- **Added vmxnet3 TX L4 checksum offload.**

  Added support for TCP/UDP checksum offload to vmxnet3.

- **Added vmxnet3 TSO support.**

  Added support for TSO to vmxnet3.

- **Added vmxnet3 support for jumbo frames.**

  Added support for linking multi-segment buffers together to handle Jumbo packets.

- **Enabled Virtio 1.0 support.**

  Enabled Virtio 1.0 support for Virtio pmd driver.

- **Supported Virtio for ARM.**

  Enabled Virtio support for ARMv7/v8. Tested for ARM64. Virtio for ARM supports VFIO-noiommu mode only. Virtio can work with other non-x86 architectures as well, like PowerPC.

- **Supported Virtio offload in vhost-user.**

  Added the offload and negotiation of checksum and TSO between vhost-user and vanilla Linux Virtio guest.

- **Added vhost-user live migration support.**

- **Added vhost driver.**

  Added a virtual PMD that wraps `librte_vhost`.

- **Added multicast promiscuous mode support on VF for ixgbe.**

  Added multicast promiscuous mode support for the ixgbe VF driver so all VFs can receive the multicast packets.

  Please note if you want to use this promiscuous mode, you need both PF and VF driver to support it. The reason is that this VF feature is configured in the PF. If you use kernel PF driver and the dpdk VF driver, make sure the kernel PF driver supports VF multicast promiscuous mode. If you use dpdk PF and dpdk VF ensure the PF driver is the same version as the VF.

- **Added support for E-tag on X550.**

  E-tag is defined in 802.1BR - Bridge Port Extension.

  This feature is for the VF, but the settings are on the PF. It means the CLIs should be used on the PF, but some of their effects will be shown on the VF. The forwarding of E-tag packets based on GRP and E-CID_base will have an effect on the PF. Theoretically, the E-tag packets can be forwarded to any pool/queue but normally we'd like to forward the packets to the pools/queues belonging to the VFs. And E-tag insertion and stripping will have an effect on VFs. When a VF receives E-tag packets it should strip the E-tag. When the VF transmits packets, it should insert the E-tag. Both actions can be offloaded.

When we want to use this E-tag support feature, the forwarding should be enabled to forward the packets received by the PF to the indicated VFs. And insertion and stripping should be enabled for VFs to offload the effort to hardware.

Features added:

  - Support E-tag offloading of insertion and stripping.
  - Support Forwarding E-tag packets to pools based on GRP and E-CID_base.

- **Added support for VxLAN and NVGRE checksum off-load on X550.**
  - Added support for VxLAN and NVGRE RX/TX checksum off-load on X550. RX/TX checksum off-load is provided on both inner and outer IP header and TCP header.
  - Added functions to support VxLAN port configuration. The default VxLAN port number is 4789 but this can be updated programmatically.
- **Added support for new X550EM_a devices.**

  Added support for new X550EM_a devices and their MAC types, X550EM_a and X550EM_a_vf. Updated the relevant PMD to use the new devices and MAC types.
- **Added x550em_x V2 device support.**

  Added support for x550em_x V2 device. Only x550em_x V1 was supported before. A mask for V1 and V2 is defined and used to support both.
- **Supported link speed auto-negotiation on X550EM_X**

  Normally the auto-negotiation is supported by firmware and software doesn't care about it. But on x550em_x, firmware doesn't support auto-negotiation. As the ports of x550em_x are 10GbE, if we connect the port with a peer which is 1GbE, the link will always be down. We added the support for auto-negotiation by software to avoid this link down issue.
- **Added software-firmware sync on X550EM_a.**

  Added support for software-firmware sync for resource sharing. Use the PHY token, shared between software-firmware for PHY access on X550EM_a.
- **Updated the i40e base driver.**

  The i40e base driver was updated with changes including the following:
  - Use RX control AQ commands to read/write RX control registers.
  - Add new X722 device IDs, and removed X710 one was never used.
  - Expose registers for HASH/FD input set configuring.
- **Enabled PCI extended tag for i40e.**

  Enabled extended tag for i40e by checking and writing corresponding PCI config space bytes, to boost the performance. The legacy method of reading/writing sysfile supported by kernel module igb_uio is now deprecated.
- **Added i40e support for setting mac addresses.**
- **Added dump of i40e registers and EEPROM.**
- **Supported ether type setting of single and double VLAN for i40e**

- **Added VMDQ DCB mode in i40e.**

  Added support for DCB in VMDQ mode to i40e driver.

- **Added i40e VEB switching support.**
- **Added Flow director enhancements in i40e.**
- **Added PF reset event reporting in i40e VF driver.**
- **Added fm10k RX interrupt support.**
- **Optimized fm10k TX.**

  Optimized fm10k TX by freeing multiple mbufs at a time.

- **Handled error flags in fm10k vector RX.**

  Parse error flags in RX descriptor and set error bits in mbuf with vector instructions.

- **Added fm10k FTAG based forwarding support.**
- **Added mlx5 flow director support.**

  Added flow director support (`RTE_FDIR_MODE_PERFECT` and `RTE_FDIR_MODE_PERFECT_MAC_VLAN`).

  Only available with Mellanox OFED >= 3.2.

- **Added mlx5 RX VLAN stripping support.**

  Added support for RX VLAN stripping.

  Only available with Mellanox OFED >= 3.2.

- **Added mlx5 link up/down callbacks.**

  Implemented callbacks to bring link up and down.

- **Added mlx5 support for operation in secondary processes.**

  Implemented TX support in secondary processes (like mlx4).

- **Added mlx5 RX CRC stripping configuration.**

  Until now, CRC was always stripped. It can now be configured.

  Only available with Mellanox OFED >= 3.2.

- **Added mlx5 optional packet padding by HW.**

  Added an option to make PCI bus transactions rounded to a multiple of a cache line size for better alignment.

  Only available with Mellanox OFED >= 3.2.

- **Added mlx5 TX VLAN insertion support.**

  Added support for TX VLAN insertion.

  Only available with Mellanox OFED >= 3.2.

- **Changed szedata2 driver type from vdev to pdev.**

  Previously szedata2 device had to be added by `--vdev` option. Now szedata2 PMD recognizes the device automatically during EAL initialization.

- **Added szedata2 functions for setting link up/down.**
- **Added szedata2 promiscuous and allmulticast modes.**
- **Added af_packet dynamic removal function.**

  An af_packet device can now be detached using the API, like other PMD devices.

- **Increased number of next hops for LPM IPv4 to 2^24.**

  The `next_hop` field has been extended from 8 bits to 24 bits for IPv4.

- **Added support of SNOW 3G (UEA2 and UIA2) for Intel Quick Assist devices.**

  Enabled support for the SNOW 3G wireless algorithm for Intel Quick Assist devices. Support for cipher-only and hash-only is also provided along with algorithm-chaining operations.

- **Added SNOW3G SW PMD.**

  A new Crypto PMD has been added, which provides SNOW 3G UEA2 ciphering and SNOW3G UIA2 hashing.

- **Added AES GCM PMD.**

  Added new Crypto PMD to support AES-GCM authenticated encryption and authenticated decryption in software.

- **Added NULL Crypto PMD**

  Added new Crypto PMD to support null crypto operations in software.

- **Improved IP Pipeline Application.**

  The following features have been added to ip_pipeline application;

  - Added CPU utilization measurement and idle cycle rate computation.
  - Added link identification support through existing port-mask option or by specifying PCI device in every LINK section in the configuration file.
  - Added load balancing support in passthrough pipeline.

- **Added IPsec security gateway example.**

  Added a new application implementing an IPsec Security Gateway.

19.17.2 Resolved Issues

Drivers

- **ethdev: Fixed overflow for 100Gbps.**

  100Gbps in Mbps (100000) was exceeding the 16-bit max value of `link_speed` in `rte_eth_link`.

- **ethdev: Fixed byte order consistency between fdir flow and mask.**

  Fixed issue in ethdev library where the structure for setting fdir's mask and flow entry was not consistent in byte ordering.

- **cxgbe: Fixed crash due to incorrect size allocated for RSS table.**

  Fixed a segfault that occurs when accessing part of port 0's RSS table that gets overwritten by subsequent port 1's part of the RSS table due to incorrect size allocated for each entry in the table.

- **cxgbe: Fixed setting wrong device MTU.**

  Fixed an incorrect device MTU being set due to the Ethernet header and CRC lengths being added twice.

- **ixgbe: Fixed zeroed VF mac address.**

  Resolved an issue where the VF MAC address is zeroed out in cases where the VF driver is loaded while the PF interface is down. The solution is to only set it when we get an ACK from the PF.

- **ixgbe: Fixed setting flow director flag twice.**

  Resolved an issue where packets were being dropped when switching to perfect filters mode.

- **ixgbe: Set MDIO speed after MAC reset.**

  The MDIO clock speed must be reconfigured after the MAC reset. The MDIO clock speed becomes invalid, therefore the driver reads invalid PHY register values. The driver now set the MDIO clock speed prior to initializing PHY ops and again after the MAC reset.

- **ixgbe: Fixed maximum number of available TX queues.**

  In IXGBE, the maximum number of TX queues varies depending on the NIC operating mode. This was not being updated in the device information, providing an incorrect number in some cases.

- **i40e: Generated MAC address for each VFs.**

  It generates a MAC address for each VFs during PF host initialization, and keeps the VF MAC address the same among different VF launch.

- **i40e: Fixed failure of reading/writing RX control registers.**

  Fixed i40e issue of failing to read/write rx control registers when under stress with traffic, which might result in application launch failure.

- **i40e: Enabled vector driver by default.**

  Previously, vector driver was disabled by default as it couldn't fill packet type info for l3fwd to work well. Now there is an option for l3fwd to analyze the packet type so the vector driver is enabled by default.

- **i40e: Fixed link info of VF.**

  Previously, the VF's link speed stayed at 10GbE and status always was up. It did not change even when the physical link's status changed. Now this issue is fixed to make VF's link info consistent with physical link.

- **mlx5: Fixed possible crash during initialization.**

  A crash could occur when failing to allocate private device context.

- **mlx5: Added port type check.**

  Added port type check to prevent port initialization on non-Ethernet link layers and to report an error.

- **mlx5: Applied VLAN filtering to broadcast and IPv6 multicast flows.**

  Prevented reception of multicast frames outside of configured VLANs.

- **mlx5: Fixed RX checksum offload in non L3/L4 packets.**

  Fixed report of bad checksum for packets of unknown type.

- **aesni_mb: Fixed wrong return value when creating a device.**

  The `cryptodev_aesni_mb_init()` function was returning the device id of the device created, instead of 0 (on success) that `rte_eal_vdev_init()` expects. This made it impossible to create more than one aesni_mb device from the command line.

- **qat: Fixed AES GCM decryption.**

  Allowed AES GCM on the cryptodev API, but in some cases gave invalid results due to incorrect IV setting.

Libraries

- **hash: Fixed CRC32c hash computation for non multiple of 4 bytes sizes.**

  Fix crc32c hash functions to return a valid crc32c value for data lengths not a multiple of 4 bytes.

- **hash: Fixed hash library to support multi-process mode.**

  Fix hash library to support multi-process mode, using a jump table, instead of storing a function pointer to the key compare function. Multi-process mode only works with the built-in compare functions, however a custom compare function (not in the jump table) can only be used in single-process mode.

- **hash: Fixed return value when allocating an existing hash table.**

  Changed the `rte_hash*_create()` functions to return `NULL` and set `rte_errno` to `EEXIST` when the object name already exists. This is the behavior described in the API documentation in the header file. The previous behavior was to return a pointer to the existing object in that case, preventing the caller from knowing if the object had to be freed or not.

- **lpm: Fixed return value when allocating an existing object.**

  Changed the `rte_lpm*_create()` functions to return `NULL` and set `rte_errno` to `EEXIST` when the object name already exists. This is the behavior described in the API documentation in the header file. The previous behavior was to return a pointer to the existing object in that case, preventing the caller from knowing if the object had to be freed or not.

- **librte_port: Fixed segmentation fault for ring and ethdev writer nodrop.**

  Fixed core dump issue on txq and swq when dropless is set to yes.

Examples

- **l3fwd-power: Fixed memory leak for non-IP packet.**

  Fixed issue in l3fwd-power where, on receiving packets of types other than IPv4 or IPv6, the mbuf was not released, and caused a memory leak.

- **l3fwd: Fixed using packet type blindly.**

  l3fwd makes use of packet type information without querying if devices or PMDs really set it. For those devices that don't set ptypes, add an option to parse it.

- **examples/vhost: Fixed frequent mbuf allocation failure.**

  The vhost-switch often fails to allocate mbuf when dequeue from vring because it wrongly calculates the number of mbufs needed.

19.17.3 API Changes

- The ethdev statistics counter `imissed` is considered to be independent of `ierrors`. All drivers are now counting the missed packets only once, i.e. drivers will not increment ierrors anymore for missed packets.
- The ethdev structure `rte_eth_dev_info` was changed to support device speed capabilities.
- The ethdev structures `rte_eth_link` and `rte_eth_conf` were changed to support the new link API.
- The functions `rte_eth_dev_udp_tunnel_add` and `rte_eth_dev_udp_tunnel_delete` have been renamed into `rte_eth_dev_udp_tunnel_port_add` and `rte_eth_dev_udp_tunnel_port_delete`.
- The `outer_mac` and `inner_mac` fields in structure `rte_eth_tunnel_filter_conf` are changed from pointer to struct in order to keep code's readability.
- The fields in ethdev structure `rte_eth_fdir_masks` were changed to be in big endian.
- A parameter `vlan_type` has been added to the function `rte_eth_dev_set_vlan_ether_type`.
- The af_packet device init function is no longer public. The device should be attached via the API.
- The LPM `next_hop` field is extended from 8 bits to 24 bits for IPv4 while keeping ABI compatibility.
- A new `rte_lpm_config` structure is used so the LPM library will allocate exactly the amount of memory which is necessary to hold application's rules. The previous ABI is kept for compatibility.
- The prototype for the pipeline input port, output port and table action handlers are updated: the pipeline parameter is added, the packets mask parameter has been either removed or made input-only.

19.17.4 ABI Changes

- The RETA entry size in `rte_eth_rss_reta_entry64` has been increased from 8-bit to 16-bit.
- The ethdev flow director structure `rte_eth_fdir_flow` structure was changed. New fields were added to extend flow director's input set.
- The cmdline buffer size has been increase from 256 to 512.

19.17.5 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
+ libethdev.so.3
  librte_acl.so.2
  librte_cfgfile.so.2
+ librte_cmdline.so.2
  librte_distributor.so.1
  librte_eal.so.2
  librte_hash.so.2
  librte_ip_frag.so.1
  librte_ivshmem.so.1
  librte_jobstats.so.1
  librte_kni.so.2
  librte_kvargs.so.1
  librte_lpm.so.2
  librte_mbuf.so.2
  librte_mempool.so.1
  librte_meter.so.1
+ librte_pipeline.so.3
  librte_pmd_bond.so.1
  librte_pmd_ring.so.2
  librte_port.so.2
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
  librte_table.so.2
  librte_timer.so.1
  librte_vhost.so.2
```

19.17.6 Tested Platforms

1. SuperMicro 1U
   - BIOS: 1.0c
   - Processor: Intel(R) Atom(TM) CPU C2758 @ 2.40GHz
2. SuperMicro 1U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
   - Onboard NIC: Intel(R) X552/X557-AT (2x10G)
     - Firmware-version: 0x800001cf
     - Device ID (PF/VF): 8086:15ad /8086:15a8
   - kernel driver version: 4.2.5 (ixgbe)
3. SuperMicro 1U
   - BIOS: 1.0a
   - Processor: Intel(R) Xeon(R) CPU E5-4667 v3 @ 2.00GHz
4. Intel(R) Server board S2600GZ

- BIOS: SE5C600.86B.02.02.0002.122320131210
- Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz

5. Intel(R) Server board W2600CR
   - BIOS: SE5C600.86B.02.01.0002.082220131453
   - Processor: Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
6. Intel(R) Server board S2600CWT
   - BIOS: SE5C610.86B.01.01.0009.060120151350
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
7. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.01.01.0005.101720141054
   - Processor: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
8. Intel(R) Server board S2600WTT
   - BIOS: SE5C610.86B.11.01.0044.090120151156
   - Processor: Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz

19.17.7 Tested NICs

1. Intel(R) Ethernet Controller X540-AT2
   - Firmware version: 0x80000389
   - Device id (pf): 8086:1528
   - Driver version: 3.23.2 (ixgbe)
2. Intel(R) 82599ES 10 Gigabit Ethernet Controller
   - Firmware version: 0x61bf0001
   - Device id (pf/vf): 8086:10fb / 8086:10ed
   - Driver version: 4.0.1-k (ixgbe)
3. Intel(R) Corporation Ethernet Connection X552/X557-AT 10GBASE-T
   - Firmware version: 0x800001cf
   - Device id (pf/vf): 8086:15ad / 8086:15a8
   - Driver version: 4.2.5 (ixgbe)
4. Intel(R) Ethernet Converged Network Adapter X710-DA4 (4x10G)
   - Firmware version: 5.02 0x80002284
   - Device id (pf/vf): 8086:1572 / 8086:154c
   - Driver version: 1.4.26 (i40e)
5. Intel(R) Ethernet Converged Network Adapter X710-DA2 (2x10G)
   - Firmware version: 5.02 0x80002282

- Device id (pf/vf): 8086:1572 / 8086:154c
- Driver version: 1.4.25 (i40e)

6. Intel(R) Ethernet Converged Network Adapter XL710-QDA1 (1x40G)
   - Firmware version: 5.02 0x80002281
   - Device id (pf/vf): 8086:1584 / 8086:154c
   - Driver version: 1.4.25 (i40e)
7. Intel(R) Ethernet Converged Network Adapter XL710-QDA2 (2X40G)
   - Firmware version: 5.02 0x80002285
   - Device id (pf/vf): 8086:1583 / 8086:154c
   - Driver version: 1.4.25 (i40e)
8. Intel(R) 82576EB Gigabit Ethernet Controller
   - Firmware version: 1.2.1
   - Device id (pf): 8086:1526
   - Driver version: 5.2.13-k (igb)
9. Intel(R) Ethernet Controller I210
   - Firmware version: 3.16, 0x80000500, 1.304.0
   - Device id (pf): 8086:1533
   - Driver version: 5.2.13-k (igb)
10. Intel(R) Corporation I350 Gigabit Network Connection
    - Firmware version: 1.48, 0x800006e7
    - Device id (pf/vf): 8086:1521 / 8086:1520
    - Driver version: 5.2.13-k (igb)
11. Intel(R) Ethernet Multi-host Controller FM10000
    - Firmware version: N/A
    - Device id (pf/vf): 8086:15d0
    - Driver version: 0.17.0.9 (fm10k)

19.18 DPDK Release 2.2

19.18.1 New Features

- **Introduce ARMv7 and ARMv8 architectures.**
  - It is now possible to build DPDK for the ARMv7 and ARMv8 platforms.
  - ARMv7 can be tested with virtual PMD drivers.
  - ARMv8 can be tested with virtual and physical PMD drivers.

- **Enabled freeing of ring.**

  A new function `rte_ring_free()` has been added to allow the user to free a ring if it was created with `rte_ring_create()`.

- **Added keepalive support to EAL and example application.**

- **Added experimental cryptodev API**

  The cryptographic processing of packets is provided as a preview with two drivers for:

  - Intel QuickAssist devices
  - Intel AES-NI multi-buffer library

  Due to its experimental state, the API may change without prior notice.

- **Added ethdev APIs for additional IEEE1588 support.**

  Added functions to read, write and adjust system time in the NIC. Added client slave sample application to demonstrate the IEEE1588 functionality.

- **Extended Statistics.**

  Defined an extended statistics naming scheme to store metadata in the name string of each statistic. Refer to the Extended Statistics section of the Programmers Guide for more details.

  Implemented the extended statistics API for the following PMDs:

  - `igb`
  - `igbvf`
  - `i40e`
  - `i40evf`
  - `fm10k`
  - `virtio`

- **Added API in ethdev to retrieve RX/TX queue information.**

  - Added the ability for the upper layer to query RX/TX queue information.
  - Added new fields in `rte_eth_dev_info` to represent information about RX/TX descriptors min/max/align numbers, per queue, for the device.

- **Added RSS dynamic configuration to bonding.**

- **Updated the e1000 base driver.**

  The e1000 base driver was updated with several features including the following:

  - Added new i218 devices
  - Allowed both ULP and EEE in Sx state
  - Initialized 88E1543 (Marvell 1543) PHY
  - Added flags to set EEE advertisement modes
  - Supported inverted format ETrackId
  - Added bit to disable packetbuffer read
  - Added defaults for i210 RX/TX PBSIZE

  - Check more errors for ESB2 init and reset
  - Check more NVM read errors
  - Return code after setting receive address register
  - Removed all NAHUM6LP_HW tags

- **Added e1000 RX interrupt support.**
- **Added igb TSO support for both PF and VF.**
- **Added RSS enhancements to Intel x550 NIC.**
  - Added support for 512 entry RSS redirection table.
  - Added support for per VF RSS redirection table.
- **Added Flow director enhancements on Intel x550 NIC.**
  - Added 2 new flow director modes on x550. One is MAC VLAN mode, the other is tunnel mode.
- **Updated the i40e base driver.**

  The i40e base driver was updated with several changes including the following:
  - Added promiscuous on VLAN support
  - Added a workaround to drop all flow control frames
  - Added VF capabilities to virtual channel interface
  - Added TX Scheduling related AQ commands
  - Added additional PCTYPES supported for FortPark RSS
  - Added parsing for CEE DCBX TLVs
  - Added FortPark specific registers
  - Added AQ functions to handle RSS Key and LUT programming
  - Increased PF reset max loop limit
- **Added i40e vector RX/TX.**
- **Added i40e RX interrupt support.**
- **Added i40e flow control support.**
- **Added DCB support to i40e PF driver.**
- **Added RSS/FD input set granularity on Intel X710/XL710.**
- **Added different GRE key length for input set on Intel X710/XL710.**
- **Added flow director support in i40e VF.**
- **Added i40e support of early X722 series.**

  Added early X722 support, for evaluation only, as the hardware is alpha.
- **Added fm10k vector RX/TX.**
- **Added fm10k TSO support for both PF and VF.**
- **Added fm10k VMDQ support.**

- **New NIC Boulder Rapid support.**

  Added support for the Boulder Rapid variant of Intel's fm10k NIC family.

- **Enhanced support for the Chelsio CXGBE driver.**

  - Added support for Jumbo Frames.
  - Optimized forwarding performance for Chelsio T5 40GbE cards.

- **Improved enic TX packet rate.**

  Reduced frequency of TX tail pointer updates to the NIC.

- **Added support for link status interrupts in mlx4.**

- **Added partial support (TX only) for secondary processes in mlx4.**

- **Added support for Mellanox ConnectX-4 adapters (mlx5).**

  The mlx5 poll-mode driver implements support for Mellanox ConnectX-4 EN and Mellanox ConnectX-4 Lx EN families of 10/25/40/50/100 Gb/s adapters.

  Like mlx4, this PMD is only available for Linux and is disabled by default due to external dependencies (libibverbs and libmlx5).

- **Added driver for Netronome nfp-6xxx card.**

  Support for using Netronome nfp-6xxx with PCI VFs.

- **Added virtual szedata2 driver for COMBO cards.**

  Added virtual PMD for COMBO-100G and COMBO-80G cards. PMD is disabled in default configuration.

- **Enhanced support for virtio driver.**

  - Virtio ring layout optimization (fixed avail ring)
  - Vector RX
  - Simple TX

- **Added vhost-user multiple queue support.**

- **Added port hotplug support to vmxnet3.**

- **Added port hotplug support to xenvirt.**

- **Added ethtool shim and sample application.**

- **Added experimental performance thread example application.**

  The new sample application demonstrates L3 forwarding with different threading models: pthreads, cgroups, or lightweight threads. The example includes a simple cooperative scheduler.

  Due to its experimental state this application may change without notice. The application is supported only for Linux x86_64.

- **Enhancements to the IP pipeline application.**

  The following features have been added to the `ip_pipeline` application;

  - Added Multiple Producers/Multiple Consumers (MPSC) and fragmentation/reassembly support to software rings.

- Added a dynamic pipeline reconfiguration feature that allows binding a pipeline to other threads at runtime using CLI commands.
- Added enable/disable of `promisc` mode from `ip_pipeline` configuration file.
- Added check on RX queues and TX queues of each link whether they are used correctly in the `ip_pipeline` configuration file.
- Added flow id parameters to the flow-classification table entries.
- Added more functions to the routing pipeline: ARP table enable/disable, Q-in-Q and MPLS encapsulation, add color (traffic-class for QoS) to the MPLS tag.
- Added flow-actions pipeline for traffic metering/marking (for e.g. Two Rate Three Color Marker (trTCM)), policer etc.
- Modified the pass-through pipeline's actions-handler to implement a generic approach to extract fields from the packet's header and copy them to packet metadata.

19.18.2 Resolved Issues

EAL

- **eal/linux: Fixed epoll timeout.**

  Fixed issue where the `rte_epoll_wait()` function didn't return when the underlying call to `epoll_wait()` timed out.

Drivers

- **e1000/base: Synchronize PHY interface on non-ME systems.**

  On power up, the MAC - PHY interface needs to be set to PCIe, even if the cable is disconnected. In ME systems, the ME handles this on exit from the Sx (Sticky mode) state. In non-ME, the driver handles it. Added a check for non-ME system to the driver code that handles it.

- **e1000/base: Increased timeout of reset check.**

  Previously, in `check_reset_block` RSPCIPHY was polled for 100 ms before determining that the ME veto was set. This was not enough and it was increased to 300 ms.

- **e1000/base: Disabled IPv6 extension header parsing on 82575.**

  Disabled IPv6 options as per hardware limitation.

- **e1000/base: Prevent ULP flow if cable connected.**

  Enabling ULP on link down when the cable is connected caused an infinite loop of link up/down indications in the NDIS driver. The driver now enables ULP only when the cable is disconnected.

- **e1000/base: Support different EEARBC for i210.**

  EEARBC has changed on i210. It means EEARBC has a different address on i210 than on other NICs. So, add a new entity named EEARBC_I210 to the register list and make sure the right one is being used on i210.

- **e1000/base: Fix K1 configuration.**

  Added fix for the following updates to the K1 configurations: TX idle period for entering K1 should be 128 ns. Minimum TX idle period in K1 should be 256 ns.

- **e1000/base: Fix link detect flow.**

  Fix link detect flow in case where auto-negotiate is not enabled, by calling `e1000_setup_copper_link_generic` instead of `e1000_phy_setup_autoneg`.

- **e1000/base: Fix link check for i354 M88E1112 PHY.**

  The `e1000_check_for_link_media_swap()` function is supposed to check PHY page 0 for copper and PHY page 1 for "other" (fiber) links. The driver switched back from page 1 to page 0 too soon, before `e1000_check_for_link_82575()` is executed and was never finding the link on the fiber (other).

  If the link is copper, as the M88E1112 page address is set to 1, it should be set back to 0 before checking this link.

- **e1000/base: Fix beacon duration for i217.**

  Fix for I217 Packet Loss issue - The Management Engine sets the FEXTNVM4 Beacon Duration incorrectly. This fix ensures that the correct value will always be set. Correct value for this field is 8 usec.

- **e1000/base: Fix TIPG for non 10 half duplex mode.**

  TIPG value is increased when setting speed to 10 half duplex to prevent packet loss. However, it was never decreased again when speed changed. This caused performance issues in the NDIS driver. Fix this to restore TIPG to default value on non 10 half duplex.

- **e1000/base: Fix reset of DH89XXCC SGMII.**

  For DH89XXCC_SGMII, a write flush leaves registers of this device trashed (0xFFFFFFFF). Add check for this device.

  Also, after both Port SW Reset and Device Reset case, the platform should wait at least 3ms before reading any registers. Remove this condition since waiting is conditionally executed only for Device Reset.

- **e1000/base: Fix redundant PHY power down for i210.**

  Bit 11 of PHYREG 0 is used to power down PHY. The use of PHYREG 16 is no longer necessary.

- **e1000/base: fix jumbo frame CRC failures.**

  Change the value of register 776.20[11:2] for jumbo mode from 0x1A to 0x1F. This is to enlarge the gap between read and write pointers in the TX FIFO.

- **e1000/base: Fix link flap on 82579.**

  Several customers have reported a link flap issue on 82579. The symptoms are random and intermittent link losses when 82579 is connected to specific switches. the Issue was root caused as an inter-operability problem between the NIC and at least some Broadcom PHYs in the Energy Efficient Ethernet wake mechanism.

  To fix the issue, we are disabling the Phase Locked Loop shutdown in 100M Low Power Idle. This solution will cause an increase of power in 100M EEE link. It may cost an additional 28mW in this specific mode.

- **igb: Fixed IEEE1588 frame identification in I210.**

  Fixed issue where the flag `PKT_RX_IEEE1588_PTP` was not being set in the Intel I210 NIC, as the EtherType in RX descriptor is in bits 8:10 of Packet Type and not in the default bits 0:2.

- **igb: Fixed VF start with PF stopped.**

  VF needs the PF interrupt support initialized even if not started.

- **igb: Fixed VF MAC address when using with DPDK PF.**

  Assign a random MAC address in VF when not assigned by PF.

- **igb: Removed CRC bytes from byte counter statistics.**

- **ixgbe: Fixed issue with X550 DCB.**

  Fixed a DCB issue with x550 where for 8 TCs (Traffic Classes), if a packet with user priority 6 or 7 was injected to the NIC, then the NIC would only put 3 packets into the queue. There was also a similar issue for 4 TCs.

- **ixgbe: Removed burst size restriction of vector RX.**

  Fixed issue where a burst size less than 32 didn't receive anything.

- **ixgbe: Fixed VF start with PF stopped.**

  VF needs the PF interrupt support initialized even if not started.

- **ixgbe: Fixed TX hang when RS distance exceeds HW limit.**

  Fixed an issue where the TX queue can hang when a lot of highly fragmented packets have to be sent. As part of that fix, `tx_rs_thresh` for ixgbe PMD is not allowed to be greater then to 32 to comply with HW restrictions.

- **ixgbe: Fixed rx error statistic counter.**

  Fixed an issue that the rx error counter of ixgbe was not accurate. The mac short packet discard count (mspdc) was added to the counter. Mac local faults and mac remote faults are removed as they do not count packets but errors, and jabber errors were removed as they are already accounted for by the CRC error counter. Finally the XEC (l3 / l4 checksum error) counter was removed due to errata, see commit 256ff05a9cae for details.

- **ixgbe: Removed CRC bytes from byte counter statistics.**

- **i40e: Fixed base driver allocation when not using first numa node.**

  Fixed i40e issue that occurred when a DPDK application didn't initialize ports if memory wasn't available on socket 0.

- **i40e: Fixed maximum of 64 queues per port.**

  Fixed an issue in i40e where it would not support more than 64 queues per port, even though the hardware actually supports it. The real number of queues may vary, as long as the total number of queues used in PF, VFs, VMDq and FD does not exceeds the hardware maximum.

- **i40e: Fixed statistics of packets.**

  Added discarding packets on VSI to the stats and rectify the old statistics.

- **i40e: Fixed issue of not freeing memzone.**

  Fixed an issue of not freeing a memzone in the call to free the memory for adminq DMA.

- **i40e: Removed CRC bytes from byte counter statistics.**
- **mlx: Fixed driver loading.**

  The mlx drivers were unable to load when built as a shared library, due to a missing symbol in the mempool library.

- **mlx4: Performance improvements.**

  Fixed bugs in TX and RX flows that improves mlx4 performance.

- **mlx4: Fixed TX loss after initialization.**
- **mlx4: Fixed scattered TX with too many segments.**
- **mlx4: Fixed memory registration for indirect mbuf data.**
- **vhost: Fixed Qemu shutdown.**

  Fixed issue with libvirt `virsh destroy` not killing the VM.

- **virtio: Fixed crash after changing link state.**

  Fixed IO permission in the interrupt handler.

- **virtio: Fixed crash when releasing queue.**

  Fixed issue when releasing null control queue.

Libraries

- **hash: Fixed memory allocation of Cuckoo Hash key table.**

  Fixed issue where an incorrect Cuckoo Hash key table size could be calculated limiting the size to 4GB.

- **hash: Fixed incorrect lookup if key is all zero.**

  Fixed issue in hash library that occurred if an all zero key was not added to the table and the key was looked up, resulting in an incorrect hit.

- **hash: Fixed thread scaling by reducing contention.**

  Fixed issue in the hash library where, using multiple cores with hardware transactional memory support, thread scaling did not work, due to the global ring that is shared by all cores.

Examples

- **l3fwd: Fixed crash with IPv6.**
- **vhost_xen: Fixed compile error.**

Other

- This release drops compatibility with Linux kernel 2.6.33. The minimum kernel requirement is now 2.6.34.

19.18.3 Known Issues

- Some drivers do not fill in the packet type when receiving. As the l3fwd example application requires this info, the i40e vector driver must be disabled to benefit of the packet type with i40e.
- Some (possibly all) VF drivers (e.g. i40evf) do not handle any PF reset events/requests in the VF driver. This means that the VF driver may not work after a PF reset in the host side. The workaround is to avoid triggering any PF reset events/requests on the host side.
- 100G link report support is missing.
- **Mellanox PMDs (mlx4 & mlx5):**
  - PMDs do not support CONFIG_RTE_BUILD_COMBINE_LIBS and CONFIG_RTE_BUILD_SHARED_LIB simultaneously.
  - There is performance degradation for small packets when the PMD is compiled with `SGE_WR_N = 4` compared to the performance when `SGE_WR_N = 1`. If scattered packets are not used it is recommended to compile the PMD with `SGE_WR_N = 1`.
  - When a Multicast or Broadcast packet is sent to the SR-IOV mlx4 VF, it is returned back to the port.
  - PMDs report "bad" L4 checksum when IP packet is received.
  - mlx5 PMD reports "bad" checksum although the packet has "good" checksum. Will be fixed in upcoming MLNX_OFED release.

19.18.4 API Changes

- The deprecated flow director API is removed. It was replaced by `rte_eth_dev_filter_ctrl()`.
- The `dcb_queue` is renamed to `dcb_tc` in following dcb configuration structures: `rte_eth_dcb_rx_conf`, `rte_eth_dcb_tx_conf`, `rte_eth_vmdq_dcb_conf`, `rte_eth_vmdq_dcb_tx_conf`.
- The `rte_eth_rx_queue_count()` function now returns "int" instead of "uint32_t" to allow the use of negative values as error codes on return.
- The function `rte_eal_pci_close_one()` is removed. It was replaced by `rte_eal_pci_detach()`.
- The deprecated ACL API `ipv4vlan` is removed.
- The deprecated hash function `rte_jhash2()` is removed. It was replaced by `rte_jhash_32b()`.
- The deprecated KNI functions are removed: `rte_kni_create()`, `rte_kni_get_port_id()` and `rte_kni_info_get()`.
- The deprecated ring PMD functions are removed: `rte_eth_ring_pair_create()` and `rte_eth_ring_pair_attach()`.
- The devargs union field `virtual` is renamed to `virt` for C++ compatibility.

19.18.5 ABI Changes

- The EAL and ethdev structures `rte_intr_handle` and `rte_eth_conf` were changed to support RX interrupt. This was already included in 2.1 under the `CONFIG_RTE_NEXT_ABI` #define.
- The ethdev flow director entries for SCTP were changed. This was already included in 2.1 under the `CONFIG_RTE_NEXT_ABI` #define.
- The ethdev flow director structure `rte_eth_fdir_flow_ext` structure was changed. New fields were added to support flow director filtering in VF.
- The size of the ethdev structure `rte_eth_hash_filter_info` is changed by adding a new element `rte_eth_input_set_conf` in a union.
- New fields `rx_desc_lim` and `tx_desc_lim` are added into `rte_eth_dev_info` structure.
- For debug builds, the functions `rte_eth_rx_burst()`, `rte_eth_tx_burst()` `rte_eth_rx_descriptor_done()` and `rte_eth_rx_queue_count()` will no longer be separate functions in the DPDK libraries. Instead, they will only be present in the `rte_ethdev.h` header file.
- The maximum number of queues per port `CONFIG_RTE_MAX_QUEUES_PER_PORT` is increased to 1024.
- The mbuf structure was changed to support the unified packet type. This was already included in 2.1 under the `CONFIG_RTE_NEXT_ABI` #define.
- The dummy malloc library is removed. The content was moved into EAL in 2.1.
- The LPM structure is changed. The deprecated field `mem_location` is removed.
- librte_table LPM: A new parameter to hold the table name will be added to the LPM table parameter structure.
- librte_table hash: The key mask parameter is added to the hash table parameter structure for 8-byte key and 16-byte key extendable bucket and LRU tables.
- librte_port: Macros to access the packet meta-data stored within the packet buffer has been adjusted to cover the packet mbuf structure.
- librte_cfgfile: Allow longer names and values by increasing the constants `CFG_NAME_LEN` and `CFG_VALUE_LEN` to 64 and 256 respectively.
- vhost: a new field enabled is added to the `vhost_virtqueue` structure.
- vhost: a new field `virt_qp_nb` is added to `virtio_net` structure, and the `virtqueue` field is moved to the end of virtio_net structure.
- vhost: a new operation `vring_state_changed` is added to `virtio_net_device_ops` structure.
- vhost: a few spaces are reserved both at `vhost_virtqueue` and `virtio_net` structure for future extension.

19.18.6 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
+ libethdev.so.2
+ librte_acl.so.2
+ librte_cfgfile.so.2
  librte_cmdline.so.1
  librte_distributor.so.1
+ librte_eal.so.2
+ librte_hash.so.2
  librte_ip_frag.so.1
  librte_ivshmem.so.1
  librte_jobstats.so.1
+ librte_kni.so.2
  librte_kvargs.so.1
+ librte_lpm.so.2
+ librte_mbuf.so.2
  librte_mempool.so.1
  librte_meter.so.1
+ librte_pipeline.so.2
  librte_pmd_bond.so.1
+ librte_pmd_ring.so.2
+ librte_port.so.2
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
+ librte_table.so.2
  librte_timer.so.1
+ librte_vhost.so.2
```

19.19 DPDK Release 2.1

19.19.1 New Features

- **Enabled cloning of indirect mbufs.**

  This feature removes a limitation of `rte_pktmbuf_attach()` which generated the warning: "mbuf we're attaching to must be direct".

  Now, when attaching to an indirect mbuf it is possible to:

  - Copy all relevant fields (address, length, offload, ...) as before.
  - Get the pointer to the mbuf that embeds the data buffer (direct mbuf), and increase the reference counter.

  When detaching the mbuf, we can now retrieve this direct mbuf as the pointer is determined from the buffer address.

- **Extended packet type support.**

  In previous releases mbuf packet types were indicated by 6 bits in the `ol_flags`. This was not enough for some supported NICs. For example i40e hardware can recognize more than 150 packet types. Not being able to identify these additional packet types limits access to hardware offload capabilities

So an extended "unified" packet type was added to support all possible PMDs. The 16 bit packet_type in the mbuf structure was changed to 32 bits and used for this purpose.

To avoid breaking ABI compatibility, the code changes for this feature are enclosed in a `RTE_NEXT_ABI` ifdef. This is enabled by default but can be turned off for ABI compatibility with DPDK R2.0.

- **Reworked memzone to be allocated by malloc and also support freeing.**

  In the memory hierarchy, memsegs are groups of physically contiguous hugepages, memzones are slices of memsegs, and malloc slices memzones into smaller memory chunks.

  This feature modifies `malloc()` so it partitions memsegs instead of memzones. Now memzones allocate their memory from the malloc heap.

  Backward compatibility with API and ABI are maintained.

  This allow memzones, and any other structure based on memzones, for example mempools, to be freed. Currently only the API from freeing memzones is supported.

- **Interrupt mode PMD.**

  This feature introduces a low-latency one-shot RX interrupt into DPDK. It also adds a polling and interrupt mode switch control example.

  DPDK userspace interrupt notification and handling mechanism is based on UIO/VFIO with the following limitations:

  - Per queue RX interrupt events are only allowed in VFIO which supports multiple MSI-X vectors.
  - In UIO, the RX interrupt shares the same vector with other interrupts. When the RX interrupt and LSC interrupt are both enabled, only the former is available.
  - RX interrupt is only implemented for the linux target.
  - The feature is only currently enabled for tow PMDs: ixgbe and igb.

- **Packet Framework enhancements.**

  Several enhancements were made to the Packet Framework:

  - A new configuration file syntax has been introduced for IP pipeline applications. Parsing of the configuration file is changed.
  - Implementation of the IP pipeline application is modified to make it more structured and user friendly.
  - Implementation of the command line interface (CLI) for each pipeline type has been moved to the separate compilation unit. Syntax of pipeline CLI commands has been changed.
  - Initialization of IP pipeline is modified to match the new parameters structure.
  - New implementation of pass-through pipeline, firewall pipeline, routing pipeline, and flow classification has been added.
  - Master pipeline with CLI interface has been added.
  - Added extended documentation of the IP Pipeline.

- **Added API for IEEE1588 timestamping.**

This feature adds an ethdev API to enable, disable and read IEEE1588/802.1AS PTP timestamps from devices that support it. The following functions were added:

- `rte_eth_timesync_enable()`
- `rte_eth_timesync_disable()`
- `rte_eth_timesync_read_rx_timestamp()`
- `rte_eth_timesync_read_tx_timestamp()`

The "ieee1588" forwarding mode in testpmd was also refactored to demonstrate the new API.

- **Added multicast address filtering.**

  Added multicast address filtering via a new ethdev function `set_mc_addr_list()`.

  This overcomes a limitation in previous releases where the receipt of multicast packets on a given port could only be enabled by invoking the `rte_eth_allmulticast_enable()` function. This method did not work for VFs in SR-IOV architectures when the host PF driver does not allow these operation on VFs. In such cases, joined multicast addresses had to be added individually to the set of multicast addresses that are filtered by the [VF] port.

- **Added Flow Director extensions.**

  Several Flow Director extensions were added such as:

  - Support for RSS and Flow Director hashes in vector RX.
  - Added Flow Director for L2 payload.

- **Added RSS hash key size query per port.**

  This feature supports querying the RSS hash key size of each port. A new field `hash_key_size` has been added in the `rte_eth_dev_info` struct for storing hash key size in bytes.

- **Added userspace ethtool support.**

  Added userspace ethtool support to provide a familiar interface for applications that manage devices via kernel-space `ethtool_op` and `net_device_op`.

  The initial implementation focuses on operations that can be implemented through existing `netdev` APIs. More operations will be supported in later releases.

- **Updated the ixgbe base driver.**

  The ixgbe base driver was updated with several changes including the following:

  - Added a new 82599 device id.
  - Added new X550 PHY ids.
  - Added SFP+ dual-speed support.
  - Added wait helper for X550 IOSF accesses.
  - Added X550em features.
  - Added X557 PHY LEDs support.
  - Commands for flow director.
  - Issue firmware command when resetting X550em.

  See the git log for full details of the ixgbe/base changes.

- **Added additional hotplug support.**

  Port hotplug support was added to the following PMDs:

  - e1000/igb.
  - ixgbe.
  - i40e.
  - fm10k.
  - ring.
  - bonding.
  - virtio.

  Port hotplug support was added to BSD.

- **Added ixgbe LRO support.**

  Added LRO support for x540 and 82599 devices.

- **Added extended statistics for ixgbe.**

  Implemented `xstats_get()` and `xstats_reset()` in dev_ops for ixgbe to expose detailed error statistics to DPDK applications.

  These will be implemented for other PMDs in later releases.

- **Added proc_info application.**

  Created a new `proc_info` application, by refactoring the existing `dump_cfg` application, to demonstrate the usage of retrieving statistics, and the new extended statistics (see above), for DPDK interfaces.

- **Updated the i40e base driver.**

  The i40e base driver was updated with several changes including the following:

  - Support for building both PF and VF driver together.
  - Support for CEE DCBX on recent firmware versions.
  - Replacement of `i40e_debug_read_register()`.
  - Rework of `i40e_hmc_get_object_va`.
  - Update of shadow RAM read/write functions.
  - Enhancement of polling NVM semaphore.
  - Enhancements on adminq init and sending asq command.
  - Update of get/set LED functions.
  - Addition of AOC phy types to case statement in get_media_type.
  - Support for iSCSI capability.
  - Setting of FLAG_RD when sending driver version to FW.

  See the git log for full details of the i40e/base changes.

- **Added support for port mirroring in i40e.**

  Enabled mirror functionality in the i40e driver.

- **Added support for i40e double VLAN, QinQ, stripping and insertion.**

  Added support to the i40e driver for offloading double VLAN (QinQ) tags to the mbuf header, and inserting double vlan tags by hardware to the packets to be transmitted. Added a new field `vlan_tci_outer` in the `rte_mbuf` struct, and new flags in `ol_flags` to support this feature.

- **Added fm10k promiscuous mode support.**

  Added support for promiscuous/allmulticast enable and disable in the fm10k PF function. VF is not supported yet.

- **Added fm10k jumbo frame support.**

  Added support for jumbo frame less than 15K in both VF and PF functions in the fm10k pmd.

- **Added fm10k mac vlan filtering support.**

  Added support for the fm10k MAC filter, only available in PF. Updated the VLAN filter to add/delete one static entry in the MAC table for each combination of VLAN and MAC address.

- **Added support for the Broadcom bnx2x driver.**

  Added support for the Broadcom NetXtreme II bnx2x driver. It is supported only on Linux 64-bit and disabled by default.

- **Added support for the Chelsio CXGBE driver.**

  Added support for the CXGBE Poll Mode Driver for the Chelsio Terminator 5 series of 10G/40G adapters.

- **Enhanced support for Mellanox ConnectX-3 driver (mlx4).**

  - Support Mellanox OFED 3.0.
  - Improved performance for both RX and TX operations.
  - Better link status information.
  - Outer L3/L4 checksum offload support.
  - Inner L3/L4 checksum offload support for VXLAN.

- **Enabled VMXNET3 vlan filtering.**

  Added support for the VLAN filter functionality of the VMXNET3 interface.

- **Added support for vhost live migration.**

  Added support to allow live migration of vhost. Without this feature, qemu will report the following error: “migrate: Migration disabled: vhost lacks VHOST_F_LOG_ALL feature”.

- **Added support for pcap jumbo frames.**

  Extended the PCAP PMD to support jumbo frames for RX and TX.

- **Added support for the TILE-Gx architecture.**

  Added support for the EZchip TILE-Gx family of SoCs.

- **Added hardware memory transactions/lock elision for x86.**

  Added the use of hardware memory transactions (HTM) on fast-path for rwlock and spinlock (a.k.a. lock elision). The methods are implemented for x86 using Restricted Transactional Memory instructions (Intel(r) Transactional Synchronization Extensions). The implementation fall-backs to the normal rwlock if HTM is not available or memory transactions fail. This is not a replacement for all rwlock usages since not all critical sections protected by locks are friendly to HTM. For example, an attempt to perform a HW I/O operation inside a hardware memory transaction always aborts the transaction since the CPU is not able to roll-back should the transaction fail. Therefore, hardware transactional locks are not advised to be used around `rte_eth_rx_burst()` and `rte_eth_tx_burst()` calls.

- **Updated Jenkins Hash function**

  Updated the version of the Jenkins Hash (jhash) function used in DPDK from the 1996 version to the 2006 version. This gives up to 35% better performance, compared to the original one.

  Note, the hashes generated by the updated version differ from the hashes generated by the previous version.

- **Added software implementation of the Toeplitz RSS hash**

  Added a software implementation of the Toeplitz hash function used by RSS. It can be used either for packet distribution on a single queue NIC or for simulating RSS computation on a specific NIC (for example after GRE header de-encapsulation).

- **Replaced the existing hash library with a Cuckoo hash implementation.**

  Replaced the existing hash library with another approach, using the Cuckoo Hash method to resolve collisions (open addressing). This method pushes items from a full bucket when a new entry must be added to it, storing the evicted entry in an alternative location, using a secondary hash function.

  This gives the user the ability to store more entries when a bucket is full, in comparison with the previous implementation.

  The API has not been changed, although new fields have been added in the `rte_hash` structure, which has been changed to internal use only.

  The main change when creating a new table is that the number of entries per bucket is now fixed, so its parameter is ignored now (it is still there to maintain the same parameters structure).

  Also, the maximum burst size in lookup_burst function hash been increased to 64, to improve performance.

- **Optimized KNI RX burst size computation.**

  Optimized KNI RX burst size computation by avoiding checking how many entries are in `kni->rx_q` prior to actually pulling them from the fifo.

- **Added KNI multicast.**

  Enabled adding multicast addresses to KNI interfaces by adding an empty callback for `set_rx_mode` (typically used for setting up hardware) so that the ioctl succeeds. This is the same thing as the Linux tap interface does.

- **Added cmdline polling mode.**

  Added the ability to process console input in the same thread as packet processing by using the `poll()` function.

- **Added VXLAN Tunnel End point sample application.**

  Added a Tunnel End point (TEP) sample application that simulates a VXLAN Tunnel Endpoint (VTEP) termination in DPDK. It is used to demonstrate the offload and filtering capabilities of Intel XL710 10/40 GbE NICsfor VXLAN packets.

- **Enabled combining of the ``-m`` and ``–no-huge`` EAL options.**

  Added option to allow combining of the `-m` and `--no-huge` EAL command line options.

  This allows user application to run as non-root but with higher memory allocations, and removes a constraint on `--no-huge` mode being limited to 64M.

19.19.2 Resolved Issues

- **acl: Fix ambiguity between test rules.**

  Some test rules had equal priority for the same category. That could cause an ambiguity in building the trie and test results.

- **acl: Fix invalid rule wildness calculation for bitmask field type.**

- **acl: Fix matching rule.**

- **acl: Fix unneeded trie splitting for subset of rules.**

  When rebuilding a trie for limited rule-set, don't try to split the rule-set even further.

- **app/testpmd: Fix crash when port id out of bound.**

  Fixed issues in testpmd where using a port greater than 32 would cause a seg fault.

  Fixes: edab33b1c01d ("app/testpmd: support port hotplug")

- **app/testpmd: Fix reply to a multicast ICMP request.**

  Set the IP source and destination addresses in the IP header of the ICMP reply.

- **app/testpmd: fix MAC address in ARP reply.**

  Fixed issue where in the `icmpecho` forwarding mode, ARP replies from testpmd contain invalid zero-filled MAC addresses.

  Fixes: 31db4d38de72 ("net: change arp header struct declaration")

- **app/testpmd: fix default flow control values.**

  Fixes: 422a20a4e62d ("app/testpmd: fix uninitialized flow control variables")

- **bonding: Fix crash when stopping inactive slave.**

- **bonding: Fix device initialization error handling.**

- **bonding: Fix initial link status of slave.**

  On Fortville NIC, link status change interrupt callback was not executed when slave in bonding was (re-)started.

- **bonding: Fix socket id for LACP slave.**

  Fixes: 46fb43683679 ("bond: add mode 4")

- **bonding: Fix device initialization error handling.**

- **cmdline: Fix small memory leak.**

  A function in `cmdline.c` had a return that did not free the buf properly.

- **config: Enable same drivers options for Linux and BSD.**

  Enabled vector ixgbe and i40e bulk alloc for BSD as it is already done for Linux.

  Fixes: 304caba12643 ("config: fix bsd options") Fixes: 0ff3324da2eb ("ixgbe: rework vector pmd following mbuf changes")

- **devargs: Fix crash on failure.**

  This problem occurred when passing an invalid PCI id to the blacklist API in devargs.

- **e1000/i40e: Fix descriptor done flag with odd address.**

- **e1000/igb: fix ieee1588 timestamping initialization.**

  Fixed issue with e1000 ieee1588 timestamp initialization. On initialization the IEEE1588 functions read the system time to set their timestamp. However, on some 1G NICs, for example, i350, system time is disabled by default and the IEEE1588 timestamp was always 0.

- **eal/bsd: Fix inappropriate header guards.**

- **eal/bsd: Fix virtio on FreeBSD.**

  Closing the `/dev/io` fd caused a SIGBUS in inb/outb instructions as the process lost the IOPL privileges once the fd is closed.

  Fixes: 8a312224bcde ("eal/bsd: fix fd leak")

- **eal/linux: Fix comments on vfio MSI.**

- **eal/linux: Fix irq handling with igb_uio.**

  Fixed an issue where the introduction of `uio_pci_generic` broke interrupt handling with igb_uio.

  Fixes: c112df6875a5 ("eal/linux: toggle interrupt for uio_pci_generic")

- **eal/linux: Fix numa node detection.**

- **eal/linux: Fix socket value for undetermined numa node.**

  Sets zero as the default value of pci device numa_node if the socket could not be determined. This provides the same default value as FreeBSD which has no NUMA support, and makes the return value of `rte_eth_dev_socket_id()` be consistent with the API description.

- **eal/ppc: Fix cpu cycle count for little endian.**

  On IBM POWER8 PPC64 little endian architecture, the definition of tsc union will be different. This fix enables the right output from `rte_rdtsc()`.

- **ethdev: Fix check of threshold for TX freeing.**

  Fixed issue where the parameter to `tx_free_thresh` was not consistent between the drivers.

- **ethdev: Fix crash if malloc of user callback fails.**

  If `rte_zmalloc()` failed in `rte_eth_dev_callback_register` then the NULL pointer would be dereferenced.

- **ethdev: Fix illegal port access.**

To obtain a detachable flag, `pci_drv` is accessed in `rte_eth_dev_is_detachable()`. However `pci_drv` is only valid if port is enabled. Fixed by checking `rte_eth_dev_is_valid_port()` first.

- **ethdev: Make tables const.**
- **ethdev: Rename and extend the mirror type.**
- **examples/distributor: Fix debug macro.**

  The macro to turn on additional debug output when the app was compiled with `-DDEBUG` was broken.

  Fixes: 07db4a975094 ("examples/distributor: new sample app")
- **examples/kni: Fix crash on exit.**
- **examples/vhost: Fix build with debug enabled.**

  Fixes: 72ec8d77ac68 ("examples/vhost: rework duplicated code")
- **fm10k: Fix RETA table initialization.**

  The fm10k driver has 128 RETA entries in 32 registers, but it only initialized the first 32 when doing multiple RX queue configurations. This fix initializes all 128 entries.
- **fm10k: Fix RX buffer size.**
- **fm10k: Fix TX multi-segment frame.**
- **fm10k: Fix TX queue cleaning after start error.**
- **fm10k: Fix Tx queue cleaning after start error.**
- **fm10k: Fix default mac/vlan in switch.**
- **fm10k: Fix interrupt fault handling.**
- **fm10k: Fix jumbo frame issue.**
- **fm10k: Fix mac/vlan filtering.**
- **fm10k: Fix maximum VF number.**
- **fm10k: Fix maximum queue number for VF.**

  Both PF and VF shared code in function `fm10k_stats_get()`. The function worked with PF, but had problems with VF since it has less queues than PF.

  Fixes: a6061d9e7075 ("fm10k: register PF driver")
- **fm10k: Fix queue disabling.**
- **fm10k: Fix switch synchronization.**
- **i40e/base: Fix error handling of NVM state update.**
- **i40e/base: Fix hardware port number for pass-through.**
- **i40e/base: Rework virtual address retrieval for lan queue.**
- **i40e/base: Update LED blinking.**
- **i40e/base: Workaround for PHY type with firmware < 4.4.**
- **i40e: Disable setting of PHY configuration.**

- **i40e: Fix SCTP flow director.**
- **i40e: Fix check of descriptor done flag.**

  Fixes: 4861cde46116 ("i40e: new poll mode driver") Fixes: 05999aab4ca6 ("i40e: add or delete flow director")
- **i40e: Fix condition to get VMDQ info.**
- **i40e: Fix registers access from big endian CPU.**
- **i40evf: Clear command when error occurs.**
- **i40evf: Fix RSS with less RX queues than TX queues.**
- **i40evf: Fix crash when setup TX queues.**
- **i40evf: Fix jumbo frame support.**
- **i40evf: Fix offload capability flags.**

  Added checksum offload capability flags which have already been supported for a long time.
- **ivshmem: Fix crash in corner case.**

  Fixed issues where depending on the configured segments it was possible to hit a segmentation fault as a result of decrementing an unsigned index with value 0.

  Fixes: 40b966a211ab ("ivshmem: library changes for mmaping using ivshmem")
- **ixgbe/base: Fix SFP probing.**
- **ixgbe/base: Fix TX pending clearing.**
- **ixgbe/base: Fix X550 CS4227 address.**
- **ixgbe/base: Fix X550 PCIe master disabling.**
- **ixgbe/base: Fix X550 check.**
- **ixgbe/base: Fix X550 init early return.**
- **ixgbe/base: Fix X550 link speed.**
- **ixgbe/base: Fix X550em CS4227 speed mode.**
- **ixgbe/base: Fix X550em SFP+ link stability.**
- **ixgbe/base: Fix X550em UniPHY link configuration.**
- **ixgbe/base: Fix X550em flow control for KR backplane.**
- **ixgbe/base: Fix X550em flow control to be KR only.**
- **ixgbe/base: Fix X550em link setup without SFP.**
- **ixgbe/base: Fix X550em mux after MAC reset.**

  Fixes: d2e72774e58c ("ixgbe/base: support X550")
- **ixgbe/base: Fix bus type overwrite.**
- **ixgbe/base: Fix init handling of X550em link down.**
- **ixgbe/base: Fix lan id before first i2c access.**
- **ixgbe/base: Fix mac type checks.**

- **ixgbe/base: Fix tunneled UDP and TCP frames in flow director.**
- **ixgbe: Check mbuf refcnt when clearing a ring.**

  The function to clear the TX ring when a port was being closed, e.g. on exit in testpmd, was not checking the mbuf refcnt before freeing it. Since the function in the vector driver to clear the ring after TX does not setting the pointer to NULL post-free, this caused crashes if mbuf debugging was turned on.
- **ixgbe: Fix RX with buffer address not word aligned.**

  Niantic HW expects the Header Buffer Address in the RXD must be word aligned.
- **ixgbe: Fix RX with buffer address not word aligned.**
- **ixgbe: Fix Rx queue reset.**

  Fix to reset vector related RX queue fields to their initial values.

  Fixes: c95584dc2b18 ("ixgbe: new vectorized functions for Rx/Tx")
- **ixgbe: Fix TSO in IPv6.**

  When TSO was used with IPv6, the generated frames were incorrect. The L4 frame was OK, but the length field of IPv6 header was not populated correctly.
- **ixgbe: Fix X550 flow director check.**
- **ixgbe: Fix check for split packets.**

  The check for split packets to be reassembled in the vector ixgbe PMD was incorrectly only checking the first 16 elements of the array instead of all 32.

  Fixes: cf4b4708a88a ("ixgbe: improve slow-path perf with vector scattered Rx")
- **ixgbe: Fix data access on big endian cpu.**
- **ixgbe: Fix flow director flexbytes offset.**

  Fixes: d54a9888267c ("ixgbe: support flexpayload configuration of flow director")
- **ixgbe: Fix number of segments with vector scattered Rx.**

  Fixes: cf4b4708a88a (ixgbe: improve slow-path perf with vector scattered Rx)
- **ixgbe: Fix offload config option name.**

  The RX_OLFLAGS option was renamed from DISABLE to ENABLE in the driver code and Linux config. It is now renamed also in the BSD config and documentation.

  Fixes: 359f106a69a9 ("ixgbe: prefer enabling olflags rather than not disabling")
- **ixgbe: Fix release queue mbufs.**

  The calculations of what mbufs were valid in the RX and TX queues were incorrect when freeing the mbufs for the vector PMD. This led to crashes due to invalid reference counts when mbuf debugging was turned on, and possibly other more subtle problems (such as mbufs being freed when in use) in other cases.

  Fixes: c95584dc2b18 ("ixgbe: new vectorized functions for Rx/Tx")
- **ixgbe: Move PMD specific fields out of base driver.**

  Move `rx_bulk_alloc_allowed` and `rx_vec_allowed` from `ixgbe_hw` to `ixgbe_adapter`.

Fixes: 01fa1d6215fa ("ixgbe: unify Rx setup")

- **ixgbe: Rename TX queue release function.**
- **ixgbevf: Fix RX function selection.**

  The logic to select ixgbe the VF RX function is different than the PF.

- **ixgbevf: Fix link status for PF up/down events.**
- **kni: Fix RX loop limit.**

  Loop processing packets dequeued from rx_q was using the number of packets requested, not how many it actually received.

- **kni: Fix ioctl in containers, like Docker.**
- **kni: Fix multicast ioctl handling.**
- **log: Fix crash after log_history dump.**
- **lpm: Fix big endian support.**
- **lpm: Fix depth small entry add.**
- **mbuf: Fix cloning with private mbuf data.**

  Added a new `priv_size` field in mbuf structure that should be initialized at mbuf pool creation. This field contains the size of the application private data in mbufs.

  Introduced new static inline functions `rte_mbuf_from_indirect()` and `rte_mbuf_to_baddr()` to replace the existing macros, which take the private size into account when attaching and detaching mbufs.

- **mbuf: Fix data room size calculation in pool init.**

  Deduct the mbuf data room size from `mempool->elt_size` and `priv_size`, instead of using an hardcoded value that is not related to the real buffer size.

  To use `rte_pktmbuf_pool_init()`, the user can either:

  – Give a NULL parameter to rte_pktmbuf_pool_init(): in this case, the private size is assumed to be 0, and the room size is `mp->elt_size - sizeof(struct rte_mbuf)`.

  – Give the `rte_pktmbuf_pool_private` filled with appropriate data_room_size and priv_size values.

- **mbuf: Fix init when private size is not zero.**

  Allow the user to use the default `rte_pktmbuf_init()` function even if the mbuf private size is not 0.

- **mempool: Add structure for object headers.**

  Each object stored in mempools are prefixed by a header, allowing for instance to retrieve the mempool pointer from the object. When debug is enabled, a cookie is also added in this header that helps to detect corruptions and double-frees.

  Introduced a structure that materializes the content of this header, and will simplify future patches adding things in this header.

- **mempool: Fix pages computation to determine number of objects.**

- **mempool: Fix returned value after counting objects.**

  Fixes: 148f963fb532 ("xen: core library changes")

- **mlx4: Avoid requesting TX completion events to improve performance.**

  Instead of requesting a completion event for each TX burst, request it on a fixed schedule once every MLX4_PMD_TX_PER_COMP_REQ (currently 64) packets to improve performance.

- **mlx4: Fix compilation as a shared library and on 32 bit platforms.**

- **mlx4: Fix possible crash on scattered mbuf allocation failure.**

  Fixes issue where failing to allocate a segment, `mlx4_rx_burst_sp()` could call `rte_pktmbuf_free()` on an incomplete scattered mbuf whose next pointer in the last segment is not set.

- **mlx4: Fix support for multiple vlan filters.**

  This fixes the "Multiple RX VLAN filters can be configured, but only the first one works" bug.

- **pcap: Fix storage of name and type in queues.**

  pcap_rx_queue/pcap_tx_queue should store it's own copy of name/type values, not the pointer to temporary allocated space.

- **pci: Fix memory leaks and needless increment of map address.**

- **pci: Fix uio mapping differences between linux and bsd.**

- **port: Fix unaligned access to metadata.**

  Fix RTE_MBUF_METADATA macros to allow for unaligned accesses to meta-data fields.

- **ring: Fix return of new port id on creation.**

- **timer: Fix race condition.**

  Eliminate problematic race condition in `rte_timer_manage()` that can lead to corruption of per-lcore pending-lists (implemented as skip-lists).

- **vfio: Fix overflow of BAR region offset and size.**

  Fixes: 90a1633b2347 ("eal/Linux: allow to map BARs with MSI-X tables")

- **vhost: Fix enqueue/dequeue to handle chained vring descriptors.**

- **vhost: Fix race for connection fd.**

- **vhost: Fix virtio freeze due to missed interrupt.**

- **virtio: Fix crash if CQ is not negotiated.**

  Fix NULL dereference if virtio control queue is not negotiated.

- **virtio: Fix ring size negotiation.**

  Negotiate the virtio ring size. The host may allow for very large rings but application may only want a smaller ring. Conversely, if the number of descriptors requested exceeds the virtio host queue size, then just silently use the smaller host size.

  This fixes issues with virtio in non-QEMU environments. For example Google Compute Engine allows up to 16K elements in ring.

- **vmxnet3: Fix link state handling.**

19.19.3 Known Issues

- When running the `vmdq` sample or `vhost` sample applications with the Intel(R) XL710 (i40e) NIC, the configuration option `CONFIG_RTE_MAX_QUEUES_PER_PORT` should be increased from 256 to 1024.
- VM power manager may not work on systems with more than 64 cores.

19.19.4 API Changes

- The order that user supplied RX and TX callbacks are called in has been changed to the order that they were added (fifo) in line with end-user expectations. The previous calling order was the reverse of this (lifo) and was counter intuitive for users. The actual API is unchanged.

19.19.5 ABI Changes

- The `rte_hash` structure has been changed to internal use only.

19.20 DPDK Release 2.0

19.20.1 New Features

- Poll-mode driver support for an early release of the PCIE host interface of the Intel(R) Ethernet Switch FM10000.
  - Basic Rx/Tx functions for PF/VF
  - Interrupt handling support for PF/VF
  - Per queue start/stop functions for PF/VF
  - Support Mailbox handling between PF/VF and PF/Switch Manager
  - Receive Side Scaling (RSS) for PF/VF
  - Scatter receive function for PF/VF
  - Reta update/query for PF/VF
  - VLAN filter set for PF
  - Link status query for PF/VF

Note: The software is intended to run on pre-release hardware and may contain unknown or unresolved defects or issues related to functionality and performance. The poll mode driver is also pre-release and will be updated to a released version post hardware and base driver release. Should the official hardware release be made between DPDK releases an updated poll-mode driver will be made available.

- Link Bonding
  - Support for adaptive load balancing (mode 6) to the link bonding library.
  - Support for registration of link status change callbacks with link bonding devices.

  - Support for slaves devices which do not support link status change interrupts in the link bonding library via a link status polling mechanism.
- PCI Hotplug with NULL PMD sample application
- ABI versioning
- x32 ABI
- Non-EAL Thread Support
- Multi-pthread Support
- Re-order Library
- ACL for AVX2
- Architecture Independent CRC Hash
- uio_pci_generic Support
- KNI Optimizations
- Vhost-user support
- Virtio (link, vlan, mac, port IO, perf)
- IXGBE-VF RSS
- RX/TX Callbacks
- Unified Flow Types
- Indirect Attached MBUF Flag
- Use default port configuration in TestPMD
- Tunnel offloading in TestPMD
- Poll Mode Driver - 40 GbE Controllers (librte_pmd_i40e)
  - Support for Flow Director
  - Support for ethertype filter
  - Support RSS in VF
  - Support configuring redirection table with different size from 1GbE and 10 GbE
  - 128/512 entries of 40GbE PF
  - 64 entries of 40GbE VF
  - Support configuring hash functions
  - Support for VXLAN packet on Intel® 40GbE Controllers
- Poll Mode Driver for Mellanox ConnectX-3 EN adapters (mlx4)

Note: This PMD is only available for Linux and is disabled by default due to external dependencies (libibverbs and libmlx4). Please refer to the NIC drivers guide for more information.

- Packet Distributor Sample Application
- Job Stats library and Sample Application.

- Enhanced Jenkins hash (jhash) library

Note: The hash values returned by the new jhash library are different from the ones returned by the previous library.

19.21 DPDK Release 1.8

19.21.1 New Features

- Link Bonding
  - Support for 802.3ad link aggregation (mode 4) and transmit load balancing (mode 5) to the link bonding library.
  - Support for registration of link status change callbacks with link bonding devices.
  - Support for slaves devices which do not support link status change interrupts in the link bonding library via a link status polling mechanism.
- Poll Mode Driver - 40 GbE Controllers (librte_pmd_i40e)
  - Support for Flow Director
  - Support for ethertype filter
  - Support RSS in VF
  - Support configuring redirection table with different size from 1GbE and 10 GbE
  - 128/512 entries of 40GbE PF
  - 64 entries of 40GbE VF
  - Support configuring hash functions
  - Support for VXLAN packet on Intel 40GbE Controllers
- Packet Distributor Sample Application

19.22 Known Issues and Limitations in Legacy Releases

This section describes known issues with the DPDK software that aren't covered in the version specific release notes sections.

19.22.1 Unit Test for Link Bonding may fail at test_tlb_tx_burst()

Description:
Unit tests will fail in `test_tlb_tx_burst()` function with error for uneven distribution of packets.

Implication:
Unit test link_bonding_autotest will fail.

Resolution/Workaround:
There is no workaround available.

Affected Environment/Platform:
Fedora 20.

Driver/Module:
Link Bonding.

19.22.2 Pause Frame Forwarding does not work properly on igb

Description:
For igb devices rte_eth_flow_ctrl_set does not work as expected. Pause frames are always forwarded on igb, regardless of the `RFCE`, `MPMCF` and `DPF` registers.

Implication:
Pause frames will never be rejected by the host on 1G NICs and they will always be forwarded.

Resolution/Workaround:
There is no workaround available.

Affected Environment/Platform:
All.

Driver/Module:
Poll Mode Driver (PMD).

19.22.3 In packets provided by the PMD, some flags are missing

Description:
In packets provided by the PMD, some flags are missing. The application does not have access to information provided by the hardware (packet is broadcast, packet is multicast, packet is IPv4 and so on).

Implication:
The `ol_flags` field in the `rte_mbuf` structure is not correct and should not be used.

Resolution/Workaround:
The application has to parse the Ethernet header itself to get the information, which is slower.

Affected Environment/Platform:
All.

Driver/Module:
Poll Mode Driver (PMD).

19.22.4 The rte_malloc library is not fully implemented

Description:
The `rte_malloc` library is not fully implemented.

Implication:
All debugging features of rte_malloc library described in architecture documentation are not yet implemented.

Resolution/Workaround:
No workaround available.

Affected Environment/Platform:
All.

Driver/Module:
`rte_malloc`.

19.22.5 HPET reading is slow

Description:
Reading the HPET chip is slow.

Implication:
An application that calls `rte_get_hpet_cycles()` or `rte_timer_manage()` runs slower.

Resolution/Workaround:
The application should not call these functions too often in the main loop. An alternative is to use the TSC register through `rte_rdtsc()` which is faster, but specific to an lcore and is a cycle reference, not a time reference.

Affected Environment/Platform:
All.

Driver/Module:
Environment Abstraction Layer (EAL).

19.22.6 HPET timers do not work on the Osage customer reference platform

Description:
HPET timers do not work on the Osage customer reference platform which includes an Intel® Xeon® processor 5500 series processor) using the released BIOS from Intel.

Implication:
On Osage boards, the implementation of the `rte_delay_us()` function must be changed to not use the HPET timer.

Resolution/Workaround:
This can be addressed by building the system with the `CONFIG_RTE_LIBEAL_USE_HPET=n` configuration option or by using the `--no-hpet` EAL option.

Affected Environment/Platform:
The Osage customer reference platform. Other vendor platforms with Intel® Xeon® processor 5500 series processors should work correctly, provided the BIOS supports HPET.

Driver/Module:
`lib/librte_eal/include/rte_cycles.h`

19.22.7 Not all variants of supported NIC types have been used in testing

Description:
The supported network interface cards can come in a number of variants with different device ID's. Not all of these variants have been tested with the DPDK.

The NIC device identifiers used during testing:

- Intel® Ethernet Controller XL710 for 40GbE QSFP+ [8086:1584]
- Intel® Ethernet Controller XL710 for 40GbE QSFP+ [8086:1583]
- Intel® Ethernet Controller X710 for 10GbE SFP+ [8086:1572]
- Intel® 82576 Gigabit Ethernet Controller [8086:10c9]
- Intel® 82576 Quad Copper Gigabit Ethernet Controller [8086:10e8]
- Intel® 82580 Dual Copper Gigabit Ethernet Controller [8086:150e]
- Intel® I350 Quad Copper Gigabit Ethernet Controller [8086:1521]
- Intel® 82599 Dual Fibre 10 Gigabit Ethernet Controller [8086:10fb]
- Intel® Ethernet Server Adapter X520-T2 [8086: 151c]
- Intel® Ethernet Controller X540-T2 [8086:1528]
- Intel® 82574L Gigabit Network Connection [8086:10d3]
- Emulated Intel® 82540EM Gigabit Ethernet Controller [8086:100e]
- Emulated Intel® 82545EM Gigabit Ethernet Controller [8086:100f]
- Intel® Ethernet Server Adapter X520-4 [8086:154a]
- Intel® Ethernet Controller I210 [8086:1533]

Implication:
Risk of issues with untested variants.

Resolution/Workaround:
Use tested NIC variants. For those supported Ethernet controllers, additional device IDs may be added to the software if required.

Affected Environment/Platform:
All.

Driver/Module:
Poll-mode drivers

19.22.8 Multi-process sample app requires exact memory mapping

Description:
The multi-process example application assumes that it is possible to map the hugepage memory to the same virtual addresses in client and server applications. Occasionally, very rarely with 64-bit, this does not occur and a client application will fail on startup. The Linux "address-space layout randomization" security feature can sometimes cause this to occur.

Implication:
A multi-process client application fails to initialize.

Resolution/Workaround:
See the "Multi-process Limitations" section in the DPDK Programmer's Guide for more information.

Affected Environment/Platform:
All.

Driver/Module:
Multi-process example application

19.22.9 Packets are not sent by the 1 GbE/10 GbE SR-IOV driver when the source MAC is not the MAC assigned to the VF NIC

Description:
The 1 GbE/10 GbE SR-IOV driver can only send packets when the Ethernet header's source MAC address is the same as that of the VF NIC. The reason for this is that the Linux `ixgbe` driver module in the host OS has its anti-spoofing feature enabled.

Implication:
Packets sent using the 1 GbE/10 GbE SR-IOV driver must have the source MAC address correctly set to that of the VF NIC. Packets with other source address values are dropped by the NIC if the application attempts to transmit them.

Resolution/Workaround:
Configure the Ethernet source address in each packet to match that of the VF NIC.

Affected Environment/Platform:
All.

Driver/Module:
1 GbE/10 GbE VF Poll Mode Driver (PMD).

19.22.10 SR-IOV drivers do not fully implement the rte_ethdev API

Description:
The SR-IOV drivers only supports the following rte_ethdev API functions:

- rte_eth_dev_configure()
- rte_eth_tx_queue_setup()
- rte_eth_rx_queue_setup()
- rte_eth_dev_info_get()

- rte_eth_dev_start()
- rte_eth_tx_burst()
- rte_eth_rx_burst()
- rte_eth_dev_stop()
- rte_eth_stats_get()
- rte_eth_stats_reset()
- rte_eth_link_get()
- rte_eth_link_get_no_wait()

Implication:
Calling an unsupported function will result in an application error.

Resolution/Workaround:
Do not use other rte_ethdev API functions in applications that use the SR-IOV drivers.

Affected Environment/Platform:
All.

Driver/Module:
VF Poll Mode Driver (PMD).

19.22.11 PMD does not work with –no-huge EAL command line parameter

Description:
Currently, the DPDK does not store any information about memory allocated by `malloc()` (for example, NUMA node, physical address), hence PMD drivers do not work when the ``--no-huge` command line parameter is supplied to EAL.

Implication:
Sending and receiving data with PMD will not work.

Resolution/Workaround:
Use huge page memory or use VFIO to map devices.

Affected Environment/Platform:
Systems running the DPDK on Linux

Driver/Module:
Poll Mode Driver (PMD).

19.22.12 Some hardware off-load functions are not supported by the VF Driver

Description:
Currently, configuration of the following items is not supported by the VF driver:

- IP/UDP/TCP checksum offload
- Jumbo Frame Receipt
- HW Strip CRC

Implication:
Any configuration for these items in the VF register will be ignored. The behavior is dependent on the current PF setting.

Resolution/Workaround:
For the PF (Physical Function) status on which the VF driver depends, there is an option item under PMD in the config file. For others, the VF will keep the same behavior as PF setting.

Affected Environment/Platform:
All.

Driver/Module:
VF (SR-IOV) Poll Mode Driver (PMD).

19.22.13 Kernel crash on IGB port unbinding

Description:
Kernel crash may occur when unbinding 1G ports from the igb_uio driver, on 2.6.3x kernels such as shipped with Fedora 14.

Implication:
Kernel crash occurs.

Resolution/Workaround:
Use newer kernels or do not unbind ports.

Affected Environment/Platform:
2.6.3x kernels such as shipped with Fedora 14

Driver/Module:
IGB Poll Mode Driver (PMD).

19.22.14 Twinpond and Ironpond NICs do not report link status correctly

Description:
Twin Pond/Iron Pond NICs do not bring the physical link down when shutting down the port.

Implication:
The link is reported as up even after issuing `shutdown` command unless the cable is physically disconnected.

Resolution/Workaround:
None.

Affected Environment/Platform:
Twin Pond and Iron Pond NICs

Driver/Module:
Poll Mode Driver (PMD).

19.22.15 Discrepancies between statistics reported by different NICs

Description:
Gigabit Ethernet devices from Intel include CRC bytes when calculating packet reception statistics regardless of hardware CRC stripping state, while 10-Gigabit Ethernet devices from Intel do so only when hardware CRC stripping is disabled.

Implication:
There may be a discrepancy in how different NICs display packet reception statistics.

Resolution/Workaround:
None

Affected Environment/Platform:
All.

Driver/Module:
Poll Mode Driver (PMD).

19.22.16 Error reported opening files on DPDK initialization

Description:
On DPDK application startup, errors may be reported when opening files as part of the initialization process. This occurs if a large number, for example, 500 or more, or if hugepages are used, due to the per-process limit on the number of open files.

Implication:
The DPDK application may fail to run.

Resolution/Workaround:
If using 2 MB hugepages, consider switching to a fewer number of 1 GB pages. Alternatively, use the `ulimit` command to increase the number of files which can be opened by a process.

Affected Environment/Platform:
All.

Driver/Module:
Environment Abstraction Layer (EAL).

19.22.17 Intel® QuickAssist Technology sample application does not work on a 32-bit OS on Shumway

Description:
The Intel® Communications Chipset 89xx Series device does not fully support NUMA on a 32-bit OS. Consequently, the sample application cannot work properly on Shumway, since it requires NUMA on both nodes.

Implication:
The sample application cannot work in 32-bit mode with emulated NUMA, on multi-socket boards.

Resolution/Workaround:
There is no workaround available.

Affected Environment/Platform:
Shumway

Driver/Module:
All.

19.22.18 Differences in how different Intel NICs handle maximum packet length for jumbo frame

Description:
10 Gigabit Ethernet devices from Intel do not take VLAN tags into account when calculating packet size while Gigabit Ethernet devices do so for jumbo frames.

Implication:
When receiving packets with VLAN tags, the actual maximum size of useful payload that Intel Gigabit Ethernet devices are able to receive is 4 bytes (or 8 bytes in the case of packets with extended VLAN tags) less than that of Intel 10 Gigabit Ethernet devices.

Resolution/Workaround:
Increase the configured maximum packet size when using Intel Gigabit Ethernet devices.

Affected Environment/Platform:
All.

Driver/Module:
Poll Mode Driver (PMD).

19.22.19 Binding PCI devices to igb_uio fails on Linux kernel 3.9 when more than one device is used

Description:
A known bug in the uio driver included in Linux kernel version 3.9 prevents more than one PCI device to be bound to the igb_uio driver.

Implication:
The Poll Mode Driver (PMD) will crash on initialization.

Resolution/Workaround:
Use earlier or later kernel versions, or apply the following patch.

Affected Environment/Platform:
Linux systems with kernel version 3.9

Driver/Module:
igb_uio module

19.22.20 GCC might generate Intel® AVX instructions for processors without Intel® AVX support

Description:
When compiling DPDK (and any DPDK app), gcc may generate Intel® AVX instructions, even when the processor does not support Intel® AVX.

Implication:
Any DPDK app might crash while starting up.

Resolution/Workaround:
Either compile using icc or set `EXTRA_CFLAGS='-O3'` prior to compilation.

Affected Environment/Platform:
Platforms which processor does not support Intel® AVX.

Driver/Module:
Environment Abstraction Layer (EAL).

19.22.21 Ethertype filter could receive other packets (non-assigned) in Niantic

Description:
On Intel® Ethernet Controller 82599EB When Ethertype filter (priority enable) was set, unmatched packets also could be received on the assigned queue, such as ARP packets without 802.1q tags or with the user priority not equal to set value. Launch the testpmd by disabling RSS and with multiply queues, then add the ethertype filter like the following and then start forwarding:

```
add_ethertype_filter 0 ethertype 0x0806 priority enable 3 queue 2 index 1
```

When sending ARP packets without 802.1q tag and with user priority as non-3 by tester, all the ARP packets can be received on the assigned queue.

Implication:
The user priority comparing in Ethertype filter cannot work probably. It is a NIC's issue due to the following: "In fact, ETQF.UP is not functional, and the information will be added in errata of 82599 and X540."

Resolution/Workaround:
None

Affected Environment/Platform:
All.

Driver/Module:
Poll Mode Driver (PMD).

19.22.22 Cannot set link speed on Intel® 40G Ethernet controller

Description:
On Intel® 40G Ethernet Controller you cannot set the link to specific speed.

Implication:
The link speed cannot be changed forcibly, though it can be configured by application.

Resolution/Workaround:
None

Affected Environment/Platform:
All.

Driver/Module:
Poll Mode Driver (PMD).

19.22.23 Devices bound to igb_uio with VT-d enabled do not work on Linux kernel 3.15-3.17

Description:
When VT-d is enabled (`iommu=pt intel_iommu=on`), devices are 1:1 mapped. In the Linux kernel unbinding devices from drivers removes that mapping which result in IOMMU errors. Introduced in Linux kernel 3.15 commit, solved in Linux kernel 3.18 commit.

Implication:
Devices will not be allowed to access memory, resulting in following kernel errors:

```
dmar: DRHD: handling fault status reg 2
dmar: DMAR:[DMA Read] Request device [02:00.0] fault addr a0c58000
DMAR:[fault reason 02] Present bit in context entry is clear
```

Resolution/Workaround:
Use earlier or later kernel versions, or avoid driver binding on boot by blacklisting the driver modules. I.e., in the case of `ixgbe`, we can pass the kernel command line option: `modprobe.blacklist=ixgbe`. This way we do not need to unbind the device to bind it to igb_uio.

Affected Environment/Platform:
Linux systems with kernel versions 3.15 to 3.17.

Driver/Module:
`igb_uio` module.

19.22.24 VM power manager may not work on systems with more than 64 cores

Description:
When using VM power manager on a system with more than 64 cores, VM(s) should not use cores 64 or higher.

Implication:
VM power manager should not be used with VM(s) that are using cores 64 or above.

Resolution/Workaround:
Do not use cores 64 or above.

Affected Environment/Platform:
Platforms with more than 64 cores.

Driver/Module:
VM power manager application.

19.22.25 DPDK may not build on some Intel CPUs using clang < 3.7.0

Description:
When compiling DPDK with an earlier version than 3.7.0 of clang, CPU flags are not detected on some Intel platforms such as Intel Broadwell/Skylake (and possibly future CPUs), and therefore compilation fails due to missing intrinsics.

Implication:
DPDK will not build when using a clang version < 3.7.0.

Resolution/Workaround:
Use clang 3.7.0 or higher, or gcc.

Affected Environment/Platform:
Platforms with Intel Broadwell/Skylake using an old clang version.

Driver/Module:
Environment Abstraction Layer (EAL).

19.22.26 The last EAL argument is replaced by the program name in argv[]

Description:
The last EAL argument is replaced by program name in `argv[]` after `eal_parse_args` is called. This is the intended behavior but it causes the pointer to the last EAL argument to be lost.

Implication:
If the last EAL argument in `argv[]` is generated by a malloc function, changing it will cause memory issues when freeing the argument.

Resolution/Workaround:
An application should not consider the value in `argv[]` as unchanged.

Affected Environment/Platform:
ALL.

Driver/Module:
Environment Abstraction Layer (EAL).

19.22.27 I40e VF may not receive packets in the promiscuous mode

Description:
Promiscuous mode is not supported by the DPDK i40e VF driver when using the i40e Linux kernel driver as host driver.

Implication:
The i40e VF does not receive packets when the destination MAC address is unknown.

Resolution/Workaround:
Use a explicit destination MAC address that matches the VF.

Affected Environment/Platform:
All.

Driver/Module:
Poll Mode Driver (PMD).

19.22.28 uio pci generic module bind failed in X710/XL710/XXV710

Description:
The `uio_pci_generic` module is not supported by XL710, since the errata of XL710 states that the Interrupt Status bit is not implemented. The errata is the item #71 from the xl710 controller spec. The hw limitation is the same as other X710/XXV710 NICs.

Implication:
When use `--bind=uio_pci_generic`, the `uio_pci_generic` module probes device and check the Interrupt Status bit. Since it is not supported by X710/XL710/XXV710, it return a *failed* value. The statement that these products don't support INTx masking, is indicated in the related linux kernel commit.

Resolution/Workaround:
Do not bind the `uio_pci_generic` module in X710/XL710/XXV710 NICs.

Affected Environment/Platform:
All.

Driver/Module:
Poll Mode Driver (PMD).

19.22.29 virtio tx_burst() function cannot do TSO on shared packets

Description:
The standard TX function of virtio driver does not manage shared packets properly when doing TSO. These packets should be read-only but the driver modifies them.

When doing TSO, the virtio standard expects that the L4 checksum is set to the pseudo header checksum in the packet data, which is different than the DPDK API. The driver patches the L4 checksum to conform to the virtio standard, but this solution is invalid when dealing with shared packets (clones), because the packet data should not be modified.

Implication:
In this situation, the shared data will be modified by the driver, potentially causing race conditions with the other users of the mbuf data.

Resolution/Workaround:
The workaround in the application is to ensure that the network headers in the packet data are not shared.

Affected Environment/Platform:
Virtual machines running a virtio driver.

Driver/Module:
Poll Mode Driver (PMD).

19.22.30 igb uio legacy mode can not be used in X710/XL710/XXV710

Description:
X710/XL710/XXV710 NICs lack support for indicating INTx is asserted via the interrupt bit in the PCI status register. Linux deleted them from INTx support table. The related commit.

Implication:
When insmod `igb_uio` with `intr_mode=legacy` and test link status interrupt. Since INTx interrupt is not supported by X710/XL710/XXV710, it will cause Input/Output error when reading file descriptor.

Resolution/Workaround:
Do not bind `igb_uio` with legacy mode in X710/XL710/XXV710 NICs, or do not use kernel version >4.7 when you bind `igb_uio` with legacy mode.

Affected Environment/Platform:
ALL.

Driver/Module:
Poll Mode Driver (PMD).

19.22.31 igb_uio can not be used when running l3fwd-power

Description:
Link Status Change(LSC) interrupt and packet receiving interrupt are all enabled in l3fwd-power APP. Because of UIO only support one interrupt, so these two kinds of interrupt need to share one, and the receiving interrupt have the higher priority, so can't get the right link status.

Implication:
When insmod `igb_uio` and running l3fwd-power APP, link status getting doesn't work properly.

Resolution/Workaround:
Use vfio-pci when LSC and packet receiving interrupt enabled.

Affected Environment/Platform:
ALL.

Driver/Module:
`igb_uio` module.

19.22.32 Linux kernel 4.10.0 iommu attribute read error

Description:
When VT-d is enabled (`iommu=pt intel_iommu=on`), reading IOMMU attributes from /sys/devices/virtual/iommu/dmarXXX/intel-iommu/cap on Linux kernel 4.10.0 error. This bug is fixed in Linux commit a7fdb6e648fb, This bug is introduced in Linux commit 39ab9555c241,

Implication:
When binding devices to VFIO and attempting to run testpmd application, testpmd (and other DPDK applications) will not initialize.

Resolution/Workaround:
Use other linux kernel version. It only happens in linux kernel 4.10.0.

Affected Environment/Platform:
ALL OS of linux kernel 4.10.0.

Driver/Module:
`vfio-pci` module.

19.22.33 Netvsc driver and application restart

Description:
The Linux kernel uio_hv_generic driver does not completely shutdown and clean up resources properly if application using Netvsc PMD exits.

Implication:
When application using Netvsc PMD is restarted it can not complete initialization handshake sequence with the host.

Resolution/Workaround:
Either reboot the guest or remove and reinsert the hv_uio_generic module.

Affected Environment/Platform:
Linux Hyper-V.

Driver/Module:
`uio_hv_generic` module.

19.22.34 PHY link up fails when rebinding i40e NICs to kernel driver

Description:
Some kernel drivers are not able to handle the link status correctly after DPDK application sets the PHY to link down.

Implication:
The link status can't be set to "up" after the NIC is rebound to the kernel driver. Before a DPDK application quits it will invoke the function `i40e_dev_stop()` which will sets the PHY to link down. Some kernel drivers may not be able to handle the link status correctly after it retakes control of the device. This is a known PHY link configuration issue in the i40e kernel driver. The fix has been addressed in the 2.7.4 rc version. So if the i40e kernel driver is < 2.7.4 and doesn't have the fix backported it will encounter this issue.

Resolution/Workaround:
First try to remove and reinsert the i40e kernel driver. If that fails reboot the system.

Affected Environment/Platform:
All.

Driver/Module:
Poll Mode Driver (PMD).

19.22.35 Restricted vdev ethdev operations supported in secondary process

Description
In current virtual device sharing model, Ethernet device data structure will be shared between primary and secondary process. Only those Ethernet device operations which based on it are workable in secondary process.

Implication
Some Ethernet device operations like device start/stop will be failed on virtual device in secondary process.

Affected Environment/Platform:
ALL.

Driver/Module:
Virtual Device Poll Mode Driver (PMD).

19.22.36 Kernel crash when hot-unplug igb_uio device while DPDK application is running

Description:
When device has been bound to igb_uio driver and application is running, hot-unplugging the device may cause kernel crash.

Reason:
When device is hot-unplugged, igb_uio driver will be removed which will destroy UIO resources. Later trying to access any uio resource will cause kernel crash.

Resolution/Workaround:
If using DPDK for PCI HW hot-unplug, prefer to bind device with VFIO instead of IGB_UIO.

Affected Environment/Platform:
ALL.

Driver/Module:
`igb_uio` module.

19.22.37 AVX-512 support disabled

Description:
`AVX-512` support has been disabled on some conditions. This shouldn't be confused with `CONFIG_RTE_ENABLE_AVX512` config option which is already disabled by default. This config option defines if `AVX-512` specific implementations of some file to be used or not. What has been disabled is compiler feature to produce `AVX-512` instructions from any source code.

On DPDK v18.11 `AVX-512` is disabled for all `GCC` builds which reported to cause a performance drop.

On DPDK v19.02 `AVX-512` disable scope is reduced to `GCC` and `binutils version 2.30` based on information accrued from the GCC community defect.

Reason:
Generated `AVX-512` code cause crash: https://bugs.dpdk.org/show_bug.cgi?id=97 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=88096

Resolution/Workaround:

- Update `binutils` to newer version than `2.30`.

OR

- Use different compiler, like `clang` for this case.

Affected Environment/Platform:
`GCC` and `binutils version 2.30`.

Driver/Module:
ALL.

19.22.38 Unsuitable IOVA mode may be picked as the default

Description
Not all kernel drivers and not all devices support all IOVA modes. EAL will attempt to pick a reasonable default based on a number of factors, but there may be cases where the default may be unsuitable (for example, hotplugging devices using *igb_uio* driver while having picked IOVA as VA mode on EAL initialization).

Implication
Some devices (hotplugged or otherwise) may not work due to incompatible IOVA mode being automatically picked by EAL.

Resolution/Workaround:
It is possible to force EAL to pick a particular IOVA mode by using the *–iova-mode* command-line parameter. If conflicting requirements are present (such as one device requiring IOVA as PA and one requiring IOVA as VA mode), there is no workaround.

Affected Environment/Platform:
Linux.

Driver/Module:
ALL.

19.23 ABI and API Deprecation

See the guidelines document for details of the *ABI policy*. API and ABI deprecation notices are to be posted here.

19.23.1 Deprecation Notices

- meson: The minimum supported version of meson for configuring and building DPDK will be increased to v0.47.1 (from 0.41) from DPDK 19.05 onwards. For those users with a version earlier than 0.47.1, an updated copy of meson can be got using the `pip`, or `pip3`, tool for downloading python packages.
- kvargs: The function `rte_kvargs_process` will get a new parameter for returning key match count. It will ease handling of no-match case.
- eal: The function `rte_eal_remote_launch` will return new error codes after read or write error on the pipe, instead of calling `rte_panic`.

- eal: The `rte_logs` struct and global symbol will be made private to remove it from the externally visible ABI and allow it to be updated in the future.
- rte_atomicNN_xxx: These APIs do not take memory order parameter. This does not allow for writing optimized code for all the CPU architectures supported in DPDK. DPDK will adopt C11 atomic operations semantics and provide wrappers using C11 atomic built-ins. These wrappers must be used for patches that need to be merged in 20.08 onwards. This change will not introduce any performance degradation.
- rte_smp_*mb: These APIs provide full barrier functionality. However, many use cases do not require full barriers. To support such use cases, DPDK will adopt C11 barrier semantics and provide wrappers using C11 atomic built-ins. These wrappers must be used for patches that need to be merged in 20.08 onwards. This change will not introduce any performance degradation.
- igb_uio: In the view of reducing the kernel dependency from the main tree, as a first step, the Technical Board decided to move `igb_uio` kernel module to the dpdk-kmods repository in the /linux/igb_uio/ directory in 20.11. Minutes of Technical Board Meeting of 2019-11-06.
- lib: will fix extending some enum/define breaking the ABI. There are multiple samples in DPDK that enum/define terminated with a `.*MAX.*` value which is used by iterators, and arrays holding these values are sized with this `.*MAX.*` value. So extending this enum/define increases the `.*MAX.*` value which increases the size of the array and depending on how/where the array is used this may break the ABI. `RTE_ETH_FLOW_MAX` is one sample of the mentioned case, adding a new flow type will break the ABI because of `flex_mask[RTE_ETH_FLOW_MAX]` array usage in following public struct hierarchy: `rte_eth_fdir_flex_conf -> rte_fdir_conf -> rte_eth_conf (in the middle)`. Need to identify this kind of usages and fix in 20.11, otherwise this blocks us extending existing enum/define. One solution can be using a fixed size array instead of `.*MAX.*` value.
- ethdev: Split the `struct eth_dev_ops` struct to hide it as much as possible will be done in 20.11. Currently the `struct eth_dev_ops` struct is accessible by the application because some inline functions, like `rte_eth_tx_descriptor_status()`, access the struct directly. The struct will be separate in two, the ops used by inline functions will be moved next to Rx/Tx burst functions, rest of the `struct eth_dev_ops` struct will be moved to header file for drivers to hide it from applications.
- ethdev: the legacy filter API, including `rte_eth_dev_filter_supported()`, `rte_eth_dev_filter_ctrl()` as well as filter types MACVLAN, ETHERTYPE, FLEXIBLE, SYN, NTUPLE, TUNNEL, FDIR, HASH and L2_TUNNEL, is superseded by the generic flow API (rte_flow) in PMDs that implement the latter. Target release for removal of the legacy API will be defined once most PMDs have switched to rte_flow.
- ethdev: Update API functions returning `void` to return `int` with negative errno values to indicate various error conditions (e.g. invalid port ID, unsupported operation, failed operation):
  - `rte_eth_dev_stop`
  - `rte_eth_dev_close`
- ethdev: New offload flags `DEV_RX_OFFLOAD_FLOW_MARK` will be added in 19.11. This will allow application to enable or disable PMDs from updating `rte_mbuf::hash::fdir`. This scheme will allow PMDs to avoid writes to `rte_mbuf` fields on Rx and thereby improve Rx performance if application wishes do so. In 19.11 PMDs will still update the field even when the offload is not enabled.
- ethdev: `rx_descriptor_done` dev_ops and `rte_eth_rx_descriptor_done` will be deprecated in 20.11 and will be removed in 21.11. Existing `rte_eth_rx_descriptor_status` and

`rte_eth_tx_descriptor_status` APIs can be used as replacement.

- traffic manager: All traffic manager API's in `rte_tm.h` were mistakenly made ABI stable in the v19.11 release. The TM maintainer and other contributors have agreed to keep the TM APIs as experimental in expectation of additional spec improvements. Therefore, all APIs in `rte_tm.h` will be marked back as experimental in v20.11 DPDK release. For more details, please see the thread.
- cryptodev: support for using IV with all sizes is added, J0 still can be used but only when IV length in following structs `rte_crypto_auth_xform`, `rte_crypto_aead_xform` is set to zero. When IV length is greater or equal to one it means it represents IV, when is set to zero it means J0 is used directly, in this case 16 bytes of J0 need to be passed.
- sched: To allow more traffic classes, flexible mapping of pipe queues to traffic classes, and subport level configuration of pipes and queues changes will be made to macros, data structures and API functions defined in "rte_sched.h". These changes are aligned to improvements suggested in the RFC https://mails.dpdk.org/archives/dev/2018-November/120035.html.
- metrics: The function `rte_metrics_init` will have a non-void return in order to notify errors instead of calling `rte_exit`.
- power: `rte_power_set_env` function will no longer return 0 on attempt to set new power environment if power environment was already initialized. In this case the function will return -1 unless the environment is unset first (using `rte_power_unset_env`). Other function usage scenarios will not change.
- python: Since the beginning of 2020, Python 2 has officially reached end-of-support: https://www.python.org/doc/sunset-python-2/. Python 2 support will be completely removed in 20.11. In 20.08, explicit deprecation warnings will be displayed when running scripts with Python 2.
- pci: Remove `RTE_KDRV_NONE` based device driver probing. In order to optimize the DPDK PCI enumeration management, `RTE_KDRV_NONE` based device driver probing will be removed in v20.08. The legacy virtio is the only consumer of `RTE_KDRV_NONE` based device driver probe scheme. The legacy virtio support will be available through the existing VFIO/UIO based kernel driver scheme. More details at https://patches.dpdk.org/patch/69351/

CHAPTER

TWENTY

FAQ

This document contains some Frequently Asked Questions that arise when working with DPDK.

20.1 What does "EAL: map_all_hugepages(): open failed: Permission denied Cannot init memory" mean?

This is most likely due to the test application not being run with sudo to promote the user to a superuser. Alternatively, applications can also be run as regular user. For more information, please refer to *DPDK Getting Started Guide*.

20.2 If I want to change the number of hugepages allocated, how do I remove the original pages allocated?

The number of pages allocated can be seen by executing the following command:

```
grep Huge /proc/meminfo
```

Once all the pages are mmapped by an application, they stay that way. If you start a test application with less than the maximum, then you have free pages. When you stop and restart the test application, it looks to see if the pages are available in the `/dev/huge` directory and mmaps them. If you look in the directory, you will see `n` number of 2M pages files. If you specified 1024, you will see 1024 page files. These are then placed in memory segments to get contiguous memory.

If you need to change the number of pages, it is easier to first remove the pages. The usertools/dpdk-setup.sh script provides an option to do this. See the "Quick Start Setup Script" section in the *DPDK Getting Started Guide* for more information.

20.3 If I execute "l2fwd -l 0-3 -m 64 -n 3 – -p 3", I get the following output, indicating that there are no socket 0 hugepages to allocate the mbuf and ring structures to?

I have set up a total of 1024 Hugepages (that is, allocated 512 2M pages to each NUMA node).

The -m command line parameter does not guarantee that huge pages will be reserved on specific sockets. Therefore, allocated huge pages may not be on socket 0. To request memory to be reserved on a specific socket, please use the –socket-mem command-line parameter instead of -m.

20.4 I am running a 32-bit DPDK application on a NUMA system, and sometimes the application initializes fine but cannot allocate memory. Why is that happening?

32-bit applications have limitations in terms of how much virtual memory is available, hence the number of hugepages they are able to allocate is also limited (1 GB size). If your system has a lot (>1 GB size) of hugepage memory, not all of it will be allocated. Due to hugepages typically being allocated on a local NUMA node, the hugepages allocation the application gets during the initialization depends on which NUMA node it is running on (the EAL does not affinitize cores until much later in the initialization process). Sometimes, the Linux OS runs the DPDK application on a core that is located on a different NUMA node from DPDK master core and therefore all the hugepages are allocated on the wrong socket.

To avoid this scenario, either lower the amount of hugepage memory available to 1 GB size (or less), or run the application with taskset affinitizing the application to a would-be master core.

For example, if your EAL coremask is 0xff0, the master core will usually be the first core in the coremask (0x10); this is what you have to supply to taskset:

```
taskset 0x10 ./l2fwd -l 4-11 -n 2
```

In this way, the hugepages have a greater chance of being allocated to the correct socket. Additionally, a `--socket-mem` option could be used to ensure the availability of memory for each socket, so that if hugepages were allocated on the wrong socket, the application simply will not start.

20.5 On application startup, there is a lot of EAL information printed. Is there any way to reduce this?

Yes, the option `--log-level=` accepts either symbolic names (or numbers):

1. emergency
2. alert
3. critical
4. error
5. warning
6. notice
7. info
8. debug

20.6 How can I tune my network application to achieve lower latency?

Traditionally, there is a trade-off between throughput and latency. An application can be tuned to achieve a high throughput, but the end-to-end latency of an average packet typically increases as a result. Similarly, the application can be tuned to have, on average, a low end-to-end latency at the cost of lower throughput.

To achieve higher throughput, the DPDK attempts to aggregate the cost of processing each packet individually by processing packets in bursts. Using the testpmd application as an example, the "burst" size can be set on the command line to a value of 32 (also the default value). This allows the application to request 32 packets at a time from the PMD. The testpmd application then immediately attempts to transmit all the packets that were received, in this case, all 32 packets. The packets are not transmitted until the tail pointer is updated on the corresponding TX queue of the network port. This behavior is desirable when tuning for high throughput because the cost of tail pointer updates to both the RX and TX queues can be spread across 32 packets, effectively hiding the relatively slow MMIO cost of writing to the PCIe* device.

However, this is not very desirable when tuning for low latency, because the first packet that was received must also wait for the other 31 packets to be received. It cannot be transmitted until the other 31 packets have also been processed because the NIC will not know to transmit the packets until the TX tail pointer has been updated, which is not done until all 32 packets have been processed for transmission.

To consistently achieve low latency even under heavy system load, the application developer should avoid processing packets in bunches. The testpmd application can be configured from the command line to use a burst value of 1. This allows a single packet to be processed at a time, providing lower latency, but with the added cost of lower throughput.

20.7 Without NUMA enabled, my network throughput is low, why?

I have a dual Intel® Xeon® E5645 processors 2.40 GHz with four Intel® 82599 10 Gigabit Ethernet NICs. Using eight logical cores on each processor with RSS set to distribute network load from two 10 GbE interfaces to the cores on each processor.

Without NUMA enabled, memory is allocated from both sockets, since memory is interleaved. Therefore, each 64B chunk is interleaved across both memory domains.

The first 64B chunk is mapped to node 0, the second 64B chunk is mapped to node 1, the third to node 0, the fourth to node 1. If you allocated 256B, you would get memory that looks like this:

```
256B buffer
Offset 0x00 - Node 0
Offset 0x40 - Node 1
Offset 0x80 - Node 0
Offset 0xc0 - Node 1
```

Therefore, packet buffers and descriptor rings are allocated from both memory domains, thus incurring QPI bandwidth accessing the other memory and much higher latency. For best performance with NUMA disabled, only one socket should be populated.

20.8 I am getting errors about not being able to open files. Why?

As the DPDK operates, it opens a lot of files, which can result in reaching the open files limits, which is set using the ulimit command or in the limits.conf file. This is especially true when using a large number (>512) of 2 MB huge pages. Please increase the open file limit if your application is not able to open files. This can be done either by issuing a ulimit command or editing the limits.conf file. Please consult Linux manpages for usage information.

20.9 VF driver for IXGBE devices cannot be initialized

Some versions of Linux IXGBE driver do not assign a random MAC address to VF devices at initialization. In this case, this has to be done manually on the VM host, using the following command:

```
ip link set <interface> vf <VF function> mac <MAC address>
```

where <interface> being the interface providing the virtual functions for example, eth0, <VF function> being the virtual function number, for example 0, and <MAC address> being the desired MAC address.

20.10 Is it safe to add an entry to the hash table while running?

Currently the table implementation is not a thread safe implementation and assumes that locking between threads and processes is handled by the user's application. This is likely to be supported in future releases.

20.11 What is the purpose of setting iommu=pt?

DPDK uses a 1:1 mapping and does not support IOMMU. IOMMU allows for simpler VM physical address translation. The second role of IOMMU is to allow protection from unwanted memory access by an unsafe device that has DMA privileges. Unfortunately, the protection comes with an extremely high performance cost for high speed NICs.

Setting `iommu=pt` disables IOMMU support for the hypervisor.

20.12 When trying to send packets from an application to itself, meaning smac==dmac, using Intel(R) 82599 VF packets are lost.

Check on register `LLE(PFVMTXSSW[n])`, which allows an individual pool to send traffic and have it looped back to itself.

20.13 Can I split packet RX to use DPDK and have an application's higher order functions continue using Linux pthread?

The DPDK's lcore threads are Linux pthreads bound onto specific cores. Configure the DPDK to do work on the same cores and run the application's other work on other cores using the DPDK's "coremask" setting to specify which cores it should launch itself on.

20.14 Is it possible to exchange data between DPDK processes and regular userspace processes via some shared memory or IPC mechanism?

Yes - DPDK processes are regular Linux/BSD processes, and can use all OS provided IPC mechanisms.

20.15 Can the multiple queues in Intel(R) I350 be used with DPDK?

I350 has RSS support and 8 queue pairs can be used in RSS mode. It should work with multi-queue DPDK applications using RSS.

20.16 How can hugepage-backed memory be shared among multiple processes?

See the Primary and Secondary examples in the *multi-process sample application*.

20.17 Why can't my application receive packets on my system with UEFI Secure Boot enabled?

If UEFI secure boot is enabled, the Linux kernel may disallow the use of UIO on the system. Therefore, devices for use by DPDK should be bound to the `vfio-pci` kernel module rather than `igb_uio` or `uio_pci_generic`.